
DIPLOMARBEIT

Herr
Thomas Pircher

**Implementierung einer Schrif-
terkennung auf Basis Neuro-
naler Netze am Beispiel einer
IBAN-Erkennung**

Mittweida, 2017

DIPLOMARBEIT

Implementierung einer Schrif- terkennung auf Basis Neuro- naler Netze am Beispiel einer IBAN-Erkennung

Autor:
Herr

Thomas Pircher

Studiengang:
Technische Informatik

Seminargruppe:
KT10WwA-F

Erstprüfer:
Prof. Dr.-Ing. Olaf Hagenbruch

Zweitprüfer:
DI Martin Gruschi

Einreichung:
Telfs, 28.9.2017

Verteidigung/Bewertung:
Mittweida, 2017

Bibliografische Beschreibung:

Pircher, Thomas:

Implementierung einer Schrifterkennung auf Basis Neuronaler Netze am Beispiel einer IBAN-Erkennung - 2017 – 83 Seiten Hauptteil, 9 Seiten Anhang, 38 Abbildungen, 2 Tabellen, 47 Code-Listings

Mittweida, Hochschule Mittweida, Fakultät für Angewandte Computer- und Biowissenschaften, Diplomarbeit, 2017

Referat:

Inhalt dieser Arbeit ist die Entwicklung einer Software zur Schrifterkennung, basierend auf Neuronalen Netzen. Dies soll am Beispiel einer IBAN-Erkennung skizziert werden.

Hierfür werden zuerst grundlegende Konzepte (künstlicher) Neuronaler Netze sowie der für die Zeichen-Extraktion notwendigen Bildvorverarbeitung besprochen.

Die Implementierung des Neuronalen Netzes erfolgt auf Basis eines speziellen Frameworks, zum direkten Vergleich wird auch eine (rudimentär funktionale) Eigenentwicklung vorgestellt.

Entsprechende Tests und Messungen zeigen Stärken und Schwächen der umgesetzten Lösung und liefern somit Aufschluss darüber, welches Potential für eine weitere Behandlung der Thematik besteht.

Inhalt

Inhalt	I
Abbildungsverzeichnis	V
Tabellenverzeichnis	VII
Codeverzeichnis.....	VIII
Abkürzungsverzeichnis	XI
1 Einleitung.....	1
1.1 <i>Motivation.....</i>	<i>1</i>
1.2 <i>Geschichtliches und Bedeutung</i>	<i>2</i>
1.3 <i>Über die Arbeit</i>	<i>2</i>
2 Bildvorverarbeitung	5
2.1 <i>Einleitung</i>	<i>5</i>
2.2 <i>Glätten</i>	<i>6</i>
2.2.1 <i>Beispiel</i>	<i>7</i>
2.3 <i>Schwellwertverfahren</i>	<i>7</i>
2.3.1 <i>Ausprägungen.....</i>	<i>8</i>
2.3.2 <i>Beispiel</i>	<i>8</i>
2.4 <i>Morphologie (Morphological Filtering).....</i>	<i>9</i>
2.4.1 <i>Erosion.....</i>	<i>9</i>
2.4.2 <i>Dilatation</i>	<i>9</i>
2.4.3 <i>Öffnen</i>	<i>10</i>
2.4.4 <i>Schließen</i>	<i>10</i>
2.5 <i>Konturen</i>	<i>11</i>
2.5.1 <i>Douglas-Peucker-Algorithmus.....</i>	<i>11</i>
2.6 <i>Zeichen-Extraktion</i>	<i>11</i>
3 Neuronale Netze	13
3.1 <i>Hintergrund</i>	<i>13</i>
3.1.1 <i>Motivation.....</i>	<i>13</i>
3.1.2 <i>Vorteile.....</i>	<i>14</i>

3.2	<i>Grundlagen</i>	15
3.2.1	Künstliches Neuron	15
3.2.2	Künstliche Neuronale Netze	16
3.3	<i>Ausprägungen</i>	18
3.3.1	Feedforward-Netze	18
3.3.2	Convolutional Neural Networks	18
3.3.3	Rekurrente Neuronale Netze	21
3.4	<i>Trainieren</i>	22
3.4.1	Fehlerbestimmung	22
3.4.2	Backpropagation	23
3.4.3	Praxis	24
3.5	<i>Gängige Probleme (und mögliche Lösungen)</i>	25
3.5.1	Initialisierungs-Werte	26
3.5.2	Lernrate	26
3.5.3	Lokale Minima	27
3.5.4	Überanpassung	28
3.6	<i>Universal Approximation Theorem</i>	29
4	Präzisierung	31
4.1	<i>Ziele (und Nicht-Ziele)</i>	31
4.2	<i>IBAN</i>	32
4.2.1	Hintergrund	32
4.2.2	Aufbau	32
4.2.3	Darstellung	33
4.2.4	Validierung	33
4.2.5	Beispiel	33
4.3	<i>Trainingsdaten</i>	34
4.3.1	MNIST-Datenbank	34
4.3.2	NIST SD-19	35
5	Implementierung	37
5.1	<i>Grundlagen</i>	37
5.1.1	Programmablauf	37
5.1.2	Entwicklungswerkzeuge	38
5.1.3	Vorbereitung der Trainingsbilder	38
5.1.4	IBAN-Prüfung	39
5.2	<i>Bildvorverarbeitung</i>	39
5.2.1	Anforderung	39
5.2.2	Ablauf	40
5.2.3	Umsetzung	40

5.2.4	Parameter-Wahl	44
5.3	<i>KNN-Eigenentwicklung</i>	44
5.3.1	Hintergrund	44
5.3.2	Ablauf	45
5.3.3	Umsetzung	45
5.3.4	Probleme	49
5.4	<i>Deeplearning4j</i>	50
5.4.1	Basis	51
5.4.2	Umsetzung	51
5.4.3	Besonderheiten	55
5.5	<i>Klassifizierung</i>	55
5.5.1	Vorbereitung	56
5.5.2	Auswertung	56
5.5.3	Interpretation	56
6	Nachweis der Funktionalität	59
6.1	<i>Vergleich Eigenentwicklung mit DL4J</i>	59
6.1.1	Konfiguration von DL4J	59
6.1.2	Hyperparameter	60
6.1.3	Ergebnis	60
6.2	<i>Vergleich klassisches FFN mit CNN</i>	61
6.2.1	Hyperparameter	61
6.2.2	Trainingsdaten	62
6.2.3	Ergebnis	63
6.3	<i>Erkennen von Handschrift</i>	64
6.3.1	Beispiel	64
6.3.2	Probleme (und mögliche Lösungsansätze)	65
6.4	<i>Erkennen von Maschinenschrift</i>	68
6.4.1	Analyse	68
7	Zusammenfassung	71
7.1	<i>Bewertung</i>	71
7.2	<i>Erkenntnisse</i>	71
7.3	<i>Ausblick</i>	72
	Literaturverzeichnis	75
	Hersteller- und Softwareverzeichnis	81
	Anhang	83

Setup	I
<i>java-iban</i>	<i>I</i>
<i>OpenCV</i>	<i>I</i>
<i>Deeplearning4J.....</i>	<i>II</i>
Historischer Abriss.....	V
<i>Maschinelles Sehen.....</i>	<i>V</i>
<i>Neuronale Netze.....</i>	<i>V</i>
Erkennen von allgemeinem Text	VII
Inhalt der CD	IX
Selbstständigkeitserklärung	

Abbildungsverzeichnis

Abbildung 1 Ablauf der Bildanalyse	5
Abbildung 2 Beispiel-Bild vor Glättung.....	7
Abbildung 3 Beispiel-Bild nach Glättung.....	7
Abbildung 4 Beispiel-Bild nach Thresholding.....	8
Abbildung 5 Erosion	9
Abbildung 6 Dilatation	10
Abbildung 7 Öffnen/Opening	10
Abbildung 8 Schließen/Closing.....	11
Abbildung 9 Verschiedene Rahmenbestimmungen	12
Abbildung 10 Neuron.....	13
Abbildung 11 Künstliches Neuron.....	15
Abbildung 12 Logistische Funktion	16
Abbildung 13 Künstliches Neuronales Netz.....	17
Abbildung 14 Convolutional Neural Network	19
Abbildung 15 Feature-Map	19
Abbildung 16 Max-Pooling mit Schrittweite 2.....	20
Abbildung 17 Rekurrentes Neuronales Netz.....	21
Abbildung 18 Fehlerfunktion mit 2 Gewichten.....	23
Abbildung 19 Mögliche Lernverläufe bei Optimierung von $E(\mathbf{w})$	27
Abbildung 20 Lokale Minima	27

Abbildung 21 Überanpassung	28
Abbildung 22 Aufbau einer österreichischen IBAN	33
Abbildung 23 MNIST-Auszug	34
Abbildung 24 NIST SD-19 Beispiel-Formular	36
Abbildung 25 Angesetzter Programm-Ablauf	37
Abbildung 26 Übersicht Bildvorverarbeitung.....	40
Abbildung 27 OpenCV: Berechnetes Rechteck liegt über Zeichen	43
Abbildung 28 Beispiel für Bild-Analyse	56
Abbildung 29 Beispiel einer händisch erfassten IBAN	65
Abbildung 30 Segmentierung einer IBAN	65
Abbildung 31 Händisch schlecht erfasste IBAN	65
Abbildung 32 Fehlerhafte Segmentierung einer händisch schlecht erfassten IBAN.....	66
Abbildung 33 Beispiele für die Ziffer 1 aus MNIST SD-19	66
Abbildung 34 Beispiele für die Ziffer 7 aus MNIST SD-19	66
Abbildung 35 Beispiel der 5. Stelle einer IBAN.....	67
Abbildung 36 IBAN in Maschinenschrift	68
Abbildung 37 Zeichenextraktion bei IBAN in Maschinenschrift	68
Abbildung 38 Handschriftlich erfasster Text	VII

Tabellenverzeichnis

Tabelle 1 Vergleich Eigenimplementierung mit DL4J.....	61
Tabelle 2 Vergleich klassisches FFN mit CNN	63

Codeverzeichnis

Listing 1 OpenCV: Import eines Bildes	41
Listing 2 OpenCV: Glätten eines Bildes	41
Listing 3 OpenCV: Binärisierung eines Bildes	41
Listing 4 OpenCV: Öffnen-Operation.....	41
Listing 5 OpenCV: Konturen-Bestimmung eines Bildes	42
Listing 6 OpenCV: Approximation der Konturenhülle	42
Listing 7 OpenCV: Nachbearbeitung der die Konturen umfassenden Rechtecks-Maße ..	44
Listing 8 OpenCV: Export eines Bildes.....	44
Listing 9 NNetwork: Definition der Trainings- und Test-Daten	46
Listing 10 NNetwork: Initialisierung der Schichten.....	46
Listing 11 NNetwork: Aufruf des Stochastischen Gradientenverfahrens.....	46
Listing 12 NNetwork: Gewichts-/Bias-Änderungen auf einzelne Batches	47
Listing 13 NNetwork: Bestimmung der Bildpixel-Grauwerte	47
Listing 14 NNetwork: Berechnung der Netz-Schicht-Ausgaben.....	47
Listing 15 NNetwork: Schreiben der Gewichts-/Bias-Änderungen	48
Listing 16 NNetwork: Netz-Erkennungsrate auf Basis der Testdaten	48
Listing 17 DL4J: KNN-Konfiguration (Auszug).....	52
Listing 18 DL4J: Variable Lernrate	52
Listing 19 DL4J: ConvolutionalLayer und SubsamplingLayer.....	52
Listing 20 DL4J: DenseLayer und OutputLayer.....	52

Codeverzeichnis	IX
Listing 21 DL4J: Schichten-Kombinierung	53
Listing 22 DL4J: Initialisierung MultiLayerNetwork.....	53
Listing 23 DL4J: Laden der Trainingsbilder	53
Listing 24 DL4J: Definition der Training-Batches	54
Listing 25 DL4J: Training des KNN.....	54
Listing 26 DL4J: Validierung des KNN.....	54
Listing 27 DL4J: Export	54
Listing 28 DL4J: Import	55
Listing 29 DL4J: Bestimmung der Klassifizierungs-Klassen-Reihenfolge des KNN	55
Listing 30 DL4J: Einlesen eines zu analysierenden Bildes	56
Listing 31 DL4J: Analyse eines Bildes.....	56
Listing 32 Kombination von Groß- und Kleinbuchstaben	57
Listing 33 IBAN-Validierung.....	58
Listing 34 DL4J: Definition eines 3-schichtigen FFN.....	59
Listing 35 DL4J: Integration der MNIST-Daten	59
Listing 36 Auswertung einer händisch erfassten IBAN ohne Syntax-Beachtung.....	65
Listing 37 Ergebnis einer erfolgreichen IBAN-Identifikation	65
Listing 38 Beispiel eines Zeichen-Interpretationsergebnisses.....	67
Listing 39 Fehlgeschlagene IBAN-Interpretation.....	68
Listing 40 java-iban Maven-Dependency.....	I
Listing 41 Laden einer nativen DLL in Java zur Laufzeit.....	I
Listing 42 DL4J Maven-Dependency.....	II
Listing 43 ND4J Maven-Dependency	II

Listing 44 SLF4J Maven-Dependencies	II
Listing 45 Cuda Maven-Dependency	II
Listing 46 Aktivierung CUDA-Unterstützung.....	III
Listing 47 Analyse-Ergebnis eines händisch erfassten Textes	VII

Abkürzungsverzeichnis

BLAS	Basic Linear Algebra Subprograms
BMP	Windows Bitmap
CNN	Convolutional Neural Network
CV	Computer Vision
DL4J	Deep Learning for Java
DLL	Dynamic Link Library
FFN	Feedforward Network
GUI	Graphical User Interface
IBAN	International Bank Account Number
JAR	Java Archive
JVM	Java Virtual Machine
KNN	Künstliches Neuronales Netz
LSTM	Long short-term memory
(M)NIST	(Modified) National Institute of Standards and Technology
ND4J	N-Dimensional Arrays for Java
PNG	Portable Network Graphics
ReLU	Rectified Linear Unit
RNN	Rekurrente Neuronale Netze
SLF4J	Simple Logging Facade for Java
TIFF	Tagged Image File Format

1 Einleitung

1.1 Motivation

Im Zuge der beruflichen Mitarbeit an der Neu-Entwicklung eines Online-Banking-Systems wurde u.a. auch die Anforderung besprochen, Zahlscheine und Rechnungen mit der Handy-Kamera abzufotografieren um auf diesem Wege Überweisungsaufträge zumindest teilweise automatisiert vorbelegen zu können. Zwar wurde kurz nach Beginn der Recherche-Arbeit zu möglichen (fix-fertigen) Lösungen die Anforderung wieder zurückgenommen, doch hatten bereits die ersten Berührungspunkte mit der Thematik mein Interesse an der dahinterliegenden Problematik (Wie lassen sich Sehen und Verstehen in einem Computerprogramm abbilden?) geweckt.

Das menschliche Gehirn vollbringt wahre Meisterleistungen. Im Alltag mag dies einem zwar nicht immer bewusst sein, spätestens beim Versuch manch (scheinbar) simple, menschliche Routinetätigkeit am Computer nachzubilden stößt man jedoch an Grenzen.

Eine dieser Tätigkeiten ist eben das Erkennen und Interpretieren von Schrift. Für einen Menschen bedarf es im Allgemeinen jahrelangen Trainings um jene kognitive Reife zu erlangen, variable Schriftbilder dem richtigen Kontext zuzuordnen. Ein Computer-Programm klassischer Prägung setzt oft ideale Bedingungen (geringe Variabilität in Bezug auf Schriftbild, Lichteinfall, etc.) voraus, um überhaupt in Ansätzen ähnliche Leistungen vollbringen zu können – was durchaus bemerkenswert ist, übersteigt doch die theoretische Rechenkapazität eines Computers jene des Menschen deutlich (vgl. [Wik-a]).

Auch stoßen klassische, statische (Programmier-)Ansätze an ihre Grenzen, wenn es gilt, Varianz abzubilden: zwar lassen sich ja noch (relativ) leicht Algorithmen entwerfen, welche zu einem fest-vorgegebenem Schriftbild z.B. Verzerrungen oder Drehungen berechnen und somit eine Zeichenerkennung ermöglichen, doch scheitern diese Ansätze zwangsweise, wenn individuelle Schriftbilder (Handschriften) in all ihrem Variantenreichtum auftreten – immerhin müsste bspw. jede Handschrift einzeln erfasst werden oder vom Programmierer allgemeine (Erkennungs-)Regeln abgeleitet werden können. Eine Aufgabe nahezu Sisyphos'schen Ausmaßes.

Neuronale Netze liefern hierzu einen Ausweg: im Gegensatz zum Ansatz, einem Computer genau zu sagen, was er zu tun hat, lernt ein solches Netz aus Beobachtung auf Basis von Trainingsdaten und leitet daraus Regeln bzw. Lösungen für konkrete Probleme ab. Eine vielversprechende Strategie um individuellen Anforderungen zu genügen.

1.2 Geschichtliches und Bedeutung

Ausgangspunkt im Folgenden ist der Bereich Maschinelles Sehen, worunter ein computergestützter Ansatz zur Lösung von Aufgabenstellungen, die sich an Fähigkeiten des menschlichen visuellen Systems orientieren, verstanden wird bzw. um es mit den Worten von David Marr, einem Forscher auf dem Gebiet Künstlicher Intelligenz, auszudrücken (s. [Mar], Seite 3):

*What does it mean to see? The plain man's answer (and Aristotle's, too)
would be, to know what is where by looking*

Einsatzgebiete finden sich viele: beginnend mit Gesichtserkennung zur Authentifizierung oder OCR-Software am PC, über Qualitäts- und Sicherungskontrollen (vgl. [TR09]) u.a. in der Fertigungstechnik und automatischen Bildbeschreibungen bei Facebook (s. [Hei16]) und Google, bis hin zu hoch-sensiblen Umgebungen wie autonome Fahrzeuge (s. [MPI]) oder medizinische Analysen von bspw. CT-Aufnahmen.

Gemeinsam haben alle Anwendungen, dass eine Orientierung am menschlichen visuellen System erfolgt, wobei neben der Automatisierung auch, idealerweise, auf eine Verbesserung der erreichbaren Ergebnisse abgezielt wird. Warum auch nicht, sind Computer doch zuverlässige und robuste Arbeitstiere.

Die Krux besteht jedoch darin, dass zwischen der Beschreibung und dem Verstehen von Szenen (bspw. Bildern und Videos) unterschieden werden muss: während ersteres noch recht gut mit Objekt-Erkennung, -Vermessung und (geometrischer) –Klassifikation erreicht werden (und gleichfalls Grundlage für Entscheidungsprozesse bilden) kann, setzt Verständnis (d.h. das Aufdecken von Zusammenhängen) unweit mehr Wissen voraus und hat sich als untrennbar mit der (interdisziplinären) Entwicklung von Künstlicher Intelligenz erwiesen.

Ein (kurzer) historischer Überblick findet sich im Anhang.

1.3 Über die Arbeit

Die vorliegende Arbeit hat zum Ziel neben einer übersichtlichen Beschreibung der theoretischen Basis schemenhaft die Implementierung einer Handschrift-Erkennung auf Basis Neuronaler Netze anhand der Erkennung einer IBAN zu skizzieren und die dabei erreichbaren Vorteile herauszuarbeiten.

Hierzu werden zuerst die Grundlagen besprochen: **Kapitel 2** präsentiert mögliche Bearbeitungs-Schritte um aus einem Bild Schriftzeichen extrahieren zu können. Anschließend liefert **Kapitel 3** einen Überblick zur Technik neuronaler Netze.

In **Kapitel 4** werden weitere Parameter der Arbeit spezifiziert: neben (Nicht-)Zielen der Arbeit und dem Aufbau einer IBAN wird auch die Herkunft der für das Trainieren des Neuronalen Netzes relevanten Trainingsdaten dargestellt.

Kapitel 5 beschreibt die konkrete Implementierung: zuerst werden die in Kapitel 2 besprochenen Bildvorverarbeitungs-Schritte umgesetzt, anschließend auf Basis der Erkenntnisse aus Kapitel 3 ein Neuronales Netz entwickelt. Dabei wird auch umrissen, was die Stolpersteine und Schwächen einer kompletten Eigenentwicklung sind, bevor die Umsetzung mit Hilfe eines speziellen Frameworks demonstriert wird.

Im folgenden **Kapitel 6** werden einzelne Benchmarks erhoben und die Effizienz der Implementierungen bewertet. Zudem werden einzelne Variationen der Umsetzung sowie Probleme beim praktischen Einsatz diskutiert.

Abschließend folgen in **Kapitel 7** ein Fazit der Arbeit sowie ein Ausblick auf mögliche Verbesserungen der Umsetzung.

Wohlgemerkt: Diese Arbeit kann nur einen (ausgewählten) Überblick auf ein sich rasant und stetig weiterentwickelndes Gebiet geben. Entsprechend stellt sie auch keinen Anspruch auf Vollständigkeit.

2 Bildvorverarbeitung

Im folgenden Kapitel werden, im Überblick, technische bzw. mathematische Grundlagen von relevanten Bild-Bearbeitungs-Schritten besprochen. Die konkrete Implementierung erfolgt in Kapitel 5.2. Die gewählten Bearbeitungsschritte orientieren sich an [Liu14], nähere Erläuterungen finden sich u.a. in [Par11].

2.1 Einleitung

Ziel der Bildvorverarbeitung ist die Extraktion einzelner Zeichen aus einem Bild in möglichst guter Qualität, was durch das (Hintereinander-)Ausführen einzelner Bearbeitungsschritte erreicht werden kann. Die extrahierten Zeichen können anschließend einzeln analysiert werden.

Der entsprechende Ablauf lautet somit:



Abbildung 1 Ablauf der Bildanalyse

Soweit die Theorie. In der Praxis erschweren verschiedene Störfaktoren die erfolgreiche Extraktion einzelner Zeichen. Schwierigkeiten ergeben sich u.a. aus

- Variationen bei Größe und Ausrichtung der Schrift
- Zeichenüberlagerung
- Unterschiedliche Darstellung semantisch gleicher Zeichen
- Bildrauschen

Diese Probleme treten bereits beim Erkennen von Schreibmaschinen- bzw. Computerschriften auf, verschärfen sich jedoch bei der Analyse von Handschriften, die in der Regel noch mehr Varianz aufweisen.

Da in der vorliegenden Arbeit das Erkennen von Handschriften zum Ziel gesetzt ist, müssen auch verschiedene Strategien zum Einsatz gebracht werden, welche sich den dargestellten Problemen widmen.

Die im folgende vorgestellten Ansätze orientieren sich an der späteren Umsetzung, d.h. einerseits an Verfahren welche im Framework OpenCV (Kapitel 5.2) zur Verfügung ste-

hen und klammern andererseits, aus praktischen Gründen, einige Probleme (z.B. Zeichenüberlagerung) aus.

2.2 Glätten

Unter Bildrauschen versteht man die Verschlechterung eines Bildes durch Störungen, die keinen Bezug zum eigentlichen Bildinhalt haben. Eine Möglichkeit dieser qualitativen Verschlechterung Herr zu werden, d.h. das Bild zu glätten, ist der Einsatz von Filtern: hierbei wird bspw. mit Hilfe der Fourier-Analyse das Frequenzspektrum eines Bildes gerechnet, aus welchem selektiv hohe Frequenzen entfernt werden.

Eine zur Fourier-Analyse äquivalente (und weniger aufwendige) Methode ist die Faltung (worunter man das Produkt von Funktionen versteht) eines Signals per Filterkern.

Gängige Filter sind linear: in diesen ist ein (berechneter) Ausgabe-Pixel-Wert $g(x, y)$ die (gewichtete) Summe von Eingabe-Pixel-Werten, d.h.

$$g(x, y) = \sum_{k, l} f(x + k, y + l) \cdot h(k, l)$$

Hierbei definiert $h(k, l)$ den Kernel, welcher als Koeffizient des Filters verstanden werden kann.

Bildlich kann man sich einen Filter als eine (endliche) Matrix von Koeffizienten vorstellen, welcher das Bild Pixel-weise abtastet und einzelne, störende, Ausreißer normalisiert¹. Als Ergebnis wirkt das Bild weichgezeichnet.

Ein einfacher (und in der Arbeit verwendeter) Filter ist der normalized-box-Filter (s. [OCV-a]) welcher das Ausgabe-Pixel mit dem Durschnitt der Pixel-Werte der umgehenden Nachbars-Pixel setzt, womit sich folgender Kernel ergibt:

$$K = \frac{1}{K_{width} * K_{height}} \cdot \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}$$

Die Stärke dieses (Tiefpass-)Filters liegt in seiner, den einheitlichen Gewichten geschuldeten, schnellen Implementierung und Ausführung.

¹ Eine interaktive Darstellung verschiedener Filter findet sich unter [Pow]

2.2.1 Beispiel

Vorlage für die beispielhafte Glättung bildet folgender Ausschnitt eines Graustufenbilds (am unteren Ende findet sich eine graue Papier-Falzlinie):



Abbildung 2 Beispiel-Bild vor Glättung

Nach Anwendung des normalized-box-Filters (mit Kernel-Größe 6x6) ergibt sich folgende Darstellung:



Abbildung 3 Beispiel-Bild nach Glättung

2.3 Schwellwertverfahren

Ein per Glättung vorbehandeltes und rauschfreieres Bild kann nun weiter segmentiert werden. Unter Segmentierung versteht man die Zuordnung einzelner Pixel in verschiedene Gruppen (Segmente).

Schwellwertverfahren sind ein hierfür probates Mittel: in einfachen Situation kann schnell entschieden werden, welche Pixel gesuchte Objekte darstellen und welche der Umgebung bzw. dem Hintergrund zuzuordnen sind.

Hierzu wird für jedes Pixel ein Merkmal bestimmt, welches die Zugehörigkeit zu einer Gruppe festlegt. In der Arbeit wird hierfür der Grauwert g eines Pixels, d.h. sein reiner Helligkeitswert, herangezogen. Bei einem RGB-Bild, d.h. bei einem Bild, welches im additiven Farbraum durch das Verhältnis der Farben Rot, Grün und Blau je Pixel bestimmt wird, kann dieser wie folgt berechnet werden:

$$g = 0,299 \cdot \text{Wert}(\text{Rot}) + 0,587 \cdot \text{Wert}(\text{Grün}) + 0,114 \cdot \text{Wert}(\text{Blau})$$

Die im ersten Augenblick seltsam anmutende Formel berücksichtigt die Farbempfindlichkeit des menschlichen Auges, womit die Darstellung farzunabhängig wird und sich allein auf die Helligkeit der Pixel beschränkt. Eine alternative Berechnung wäre die Bestimmung des Mittelwerts der 3 Farbwerte.

Unabhängig von der Berechnungsmethode nimmt g einen Wert zwischen 0 (Schwarz) und 255 (Weiß) ein, da die einzelnen Farb-Komponenten ebenfalls zwischen 0 und 255 (8-Bit) liegen.

Das Schwellwertverfahren bedient sich nun folgender (allgemeiner) Abbildung T , angewandt auf jeden Bild-Pixel:

$$T(g) = \begin{cases} 0, & g \leq t \\ \text{maxValue}, & g > t \end{cases}$$

Hierbei bezeichnet t den Schwellwert (von englisch „threshold“), maxValue definiert den zu setzenden Wert, sofern der Grauwert größer dem Schwellwert ist (ansonsten wird 0 gesetzt). t und maxValue nehmen Werte zwischen 0 und 255 ein.

Eine invertierte Anwendung setzt maxValue dann, falls $g \leq t$, womit die Farben Schwarz und Weiß getauscht werden. In beiden Fällen ist das Ergebnis jedoch ein Binärbild.

2.3.1 Ausprägungen

Leicht lässt sich nachvollziehen, dass das grundlegende Prinzip des Schwellwertverfahrens auf verschiedene Arten angewandt werden kann: Neben der globalen Definition eines einzelnen Schwellwerts wie oben beschrieben lassen sich auch mehrere Schwellwerte global festlegen, wodurch eine Segmentierung in mehr als 2 Segmenten möglich wird – was insbesondere bei kontrastarmen Bildern hilfreich ist.

Ebenfalls können verschiedene Schwellwerte in verschiedenen Bild-Bereichen gelten (lokales Schwellwertverfahren). Anfällig ist dieses Verfahren jedoch gegenüber einem Versatz an Grenzen der Regionen.

Eine entsprechende Weiterentwicklung (dynamisches Schwellwertverfahren) umgeht diese Sprünge, indem für jeden Pixel die Nachbarschaft betrachtet und anhand dieser der Schwellwert berechnet wird, erkaufte sich dies jedoch mit einem deutlich ansteigendem Rechenaufwand.

2.3.2 Beispiel

Die Anwendung eines (invertierten) Schwellwert-Verfahrens (mit Schwellwert 200) auf dem zuvor geglätteten Beispiel-Bild führt zu folgendem Ergebnis:



Abbildung 4 Beispiel-Bild nach Thresholding

Deutlich ist erkennbar, dass bspw. die im Ursprungs-Bild noch ersichtliche Falzlinie verschwunden ist.

2.4 Morphologie (Morphological Filtering)

Auf einem in den vorherigen Schritten ermitteltem Binärbild können nun weitere Verfahren angewandt werden, um die Objekt-Struktur in einem Bild zu verdeutlichen bzw. irrelevantes (weiter) zu eliminieren.

Grundlegende Operationen hierbei sind Erosion und Dilatation, sowie darauf bauend die kombinierten Varianten Öffnen (Opening) bzw. Schließen (Closing).

2.4.1 Erosion

Unter einer Erosion (lat. erodere = „abnagen“) versteht man das Entfernen eines Bild-Objekt-Rahmens mit Hilfe einer Strukturmaske, wodurch sich Bild-Rauschen bzw. Bild-Artefakte eliminieren lassen.

Hierbei wird für die Strukturmaske ein Bezugspunkt definiert, mit Hilfe dessen die Maske pixelweise über das Gesamtbild verschoben wird. Wird die Maske vollständig vom segmentierten Bild-Bereich bedeckt, gehört der Pixel zur neuen, erodierten Menge.

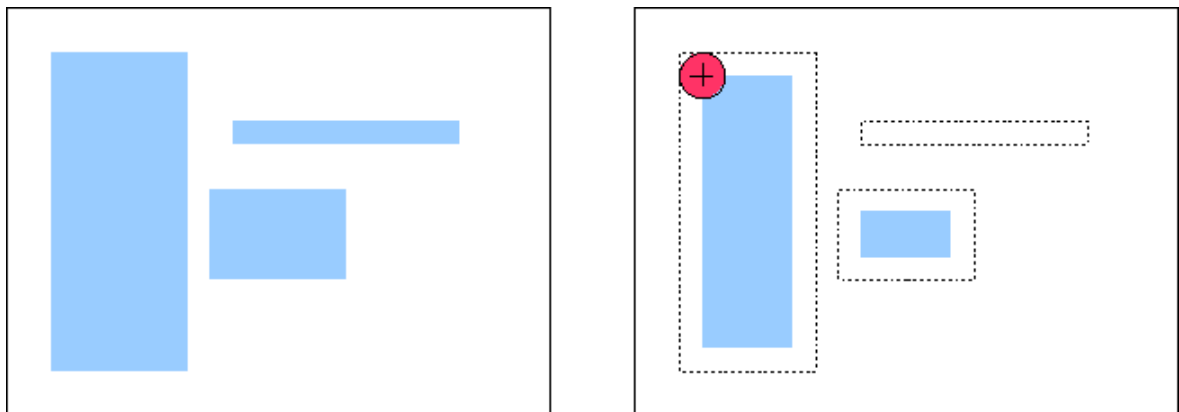


Abbildung 5 Erosion

Ist A das Bild und X die Strukturmaske, wird die Erosion mit $A \ominus X$ notiert.

2.4.2 Dilatation

Die Dilatation (lat. dilatare = „ausdehnen“) kann als Gegenteil einer Erosion gesehen werden: Gleichsam wird eine Strukturmaske X definiert, welche pixelweise über das Bild A verschoben wird, die neue Menge bildet sich nun aber aus der Vereinigung (Minkowski-Summe).

Damit wird die Dicke von Segmenten verstärkt bzw. ist die (Wieder-)Vereinigung von getrennten, jedoch zusammengehörigen Segmenten möglich.

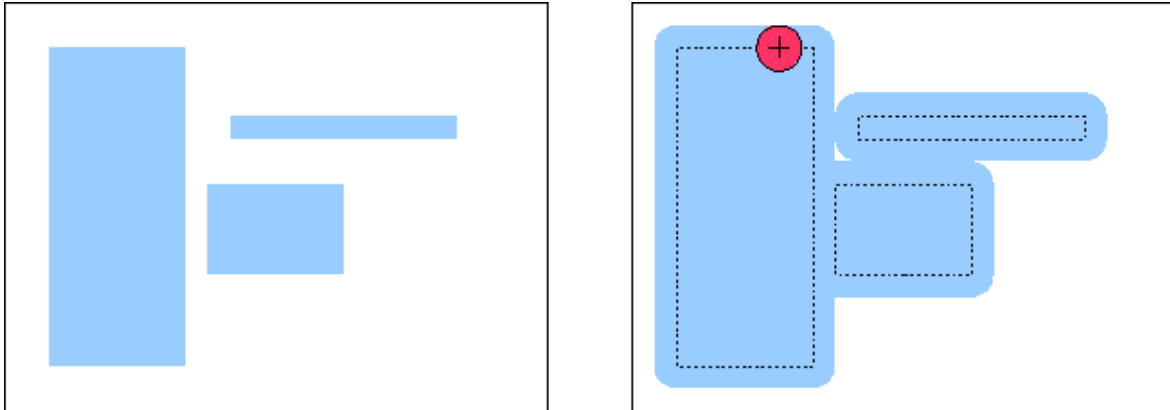


Abbildung 6 Dilation

Die Notation erfolgt mit $A \oplus X$.

2.4.3 Öffnen

Unter Öffnen (Opening) versteht man die kombinierte Anwendung von Erosion und Dilation, gegeben durch $A \circ X = (A \ominus X) \oplus X$:

Zuerst werden per Erosion alle Strukturen kleiner dem Struktur-Element X gelöscht, anschließend per Dilation die Erosion für den verbleibenden Rest wieder rückgängig gemacht.

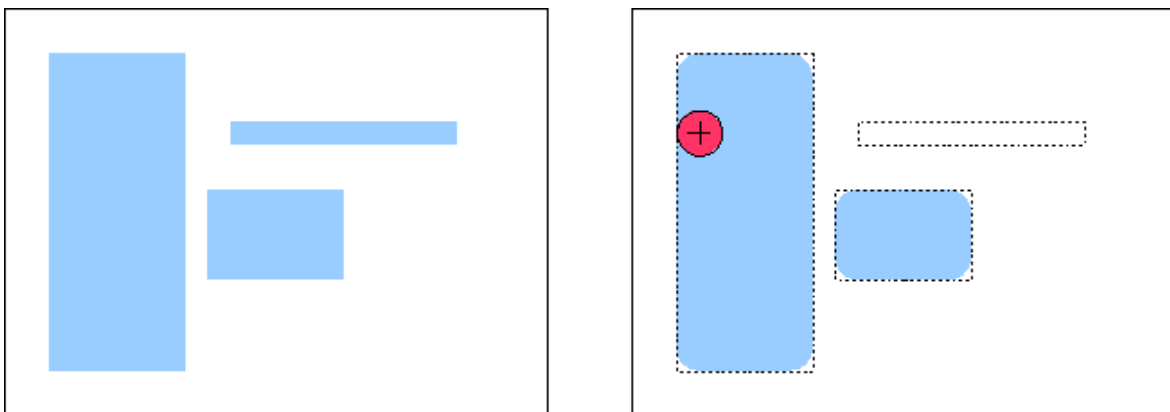


Abbildung 7 Öffnen/Opening

Als Ergebnis werden somit störende Elemente außerhalb (relevanter) Objekte entfernt.

2.4.4 Schließen

Das Schließen (Closing), definiert als $A \cdot X = (A \oplus X) \ominus X$, kann als Komplementär zum Öffnen gesehen werden:

Dilatation schließt zuerst alle Löcher, Erosion führt anschließend das Bild wieder möglichst nahe an das Original ran.

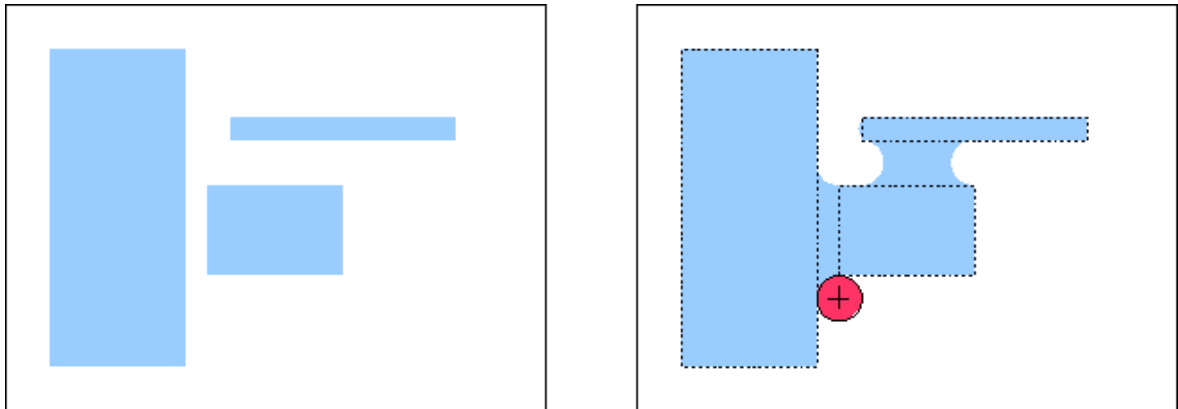


Abbildung 8 Schließen/Closing

Vollständig geschlossene Löcher bleiben damit weiterhin geschlossen, wodurch sich (bspw. rauschbedingte) Fehler beheben lassen.

2.5 Konturen

Nach Anwendung der bisherigen Bearbeitungs-Schritte sollte ein Bild vorliegen, welches nur mehr relevante Segmente umfasst – konkret in der Arbeit wären dies die Stellen einer IBAN, d.h. alphanummerische Zeichen.

Diese Zeichen gilt es nun einzeln zu extrahieren, weshalb die Umrisse (Konturen) der Segmente bestimmt werden. Konturen können hierbei als Kurven erklärt werden, welche alle durchgehenden Punkte entlang einer Grenze gleicher Farbe oder Intensität verbindet – nachdem es sich beim Ausgangsbild um ein Binärbild handelt, lassen sich die entsprechenden Grenzbereich relativ leicht klassifizieren.

2.5.1 Douglas-Peucker-Algorithmus

In einem weiteren Schritt kann bspw. mit Hilfe des Douglas-Peucker-Algorithmus eine beliebige Näherung (Simplifizierung) der Konturen-Kurve bestimmt werden. Damit sind weniger Punkte zum Beschreiben einer Kontur notwendig (vgl. [Wik-b]).

2.6 Zeichen-Extraktion

Nachdem die Segment-Konturen bestimmt sind, kann die entsprechende Info dazu verwendet werden, die einzelnen Zeichen zu extrahieren.

Je nach Anforderung können hierbei verschiedene Ansätze gewählt werden, welche sich in Form (Rechteck, Kreis, Ellipse), Ausrichtung und umschließender Fläche (die bspw. minimal sein könnte) unterscheiden (vgl. [OCV-b]).

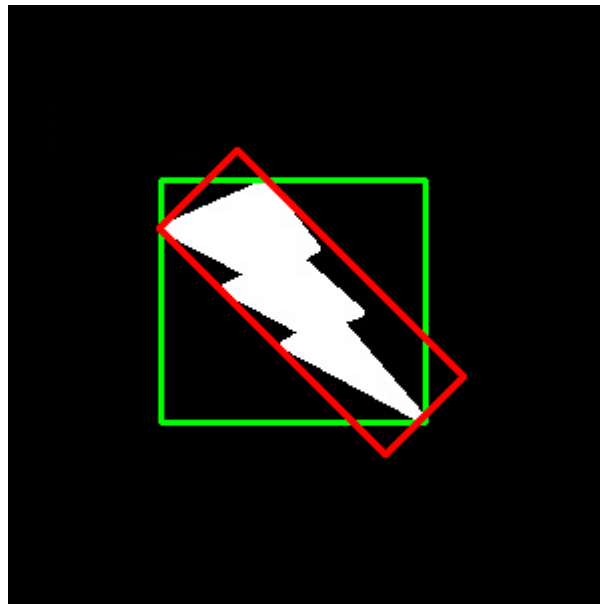


Abbildung 9 Verschiedene Rahmenbestimmungen

In der Arbeit wird ein gerades, umschließendes Rechteck verwendet (grüne Umrahmung), dessen Maße mit den Koordinaten der linken oberen Ecke sowie Breite und Höhe des Rechtecks bestimmt sind.

Die Koordinaten (x, y) der linken oberen Ecke P lassen sich wie folgt bestimmen:

$$P(x, y) = (\min(K_x), \min(K_y))$$

Hierbei sind K_x und K_y alle x- bzw. y-Werte der umschließenden Kontur K , ausgehend vom linken oberen Punkt. Die Breite w und Höhe h ergibt sich aus

$$w = \max(K_x) - x$$

sowie

$$h = \max(K_y) - y$$

Im Gegensatz zu einem umschließenden, rotiertem Rechteck (roter Rahmen) ist die Rechteck-Fläche nicht minimal, was jedoch insbesondere für den Fall, dass das relevante Zeichen orthogonal im Bild steht zu vernachlässigen ist.

Das Ergebnis ist somit eine Liste von Binärbildern mit jeweils genau einem, in weiterer Folge zu analysierendem Zeichen.

3 Neuronale Netze

Im folgenden Kapitel werden die Grundlagen zu Neuronalen Netzen erörtert sowie einige Ausprägungen vorgestellt. Ebenso werden häufige Probleme (sowie entsprechende Lösungsansätze) bei der Implementierung analysiert. Teile der folgenden Beschreibung stammen aus [Bis06], [ct16], [KrD], [Nie] und [Par11].

3.1 Hintergrund

3.1.1 Motivation

Die Theorie Neuronaler Netze fußt auf den Erkenntnissen der Gehirnforschung und versucht das komplexe Zusammenspiel zwischen Nervenzellen (Neuronen) und deren Verbindungen (Synapsen) zu modellieren.

Das menschliche Gehirn besteht aus etwa 86 Milliarden (vgl. [Her12]), auf Erregungsleitung und –übertragung spezialisierte, Neuronen, wobei jedes Neuron über Synapsen mit bis zu Tausenden anderen Neuronen verbunden ist. Während die Signale innerhalb einer Zelle elektronisch übertragen werden, erfolgt die Kommunikation zwischen Zellen mit Hilfe chemischer Botenstoffe (Neurotransmitter).

Je näher eine Synapse am Zellkörper (Soma) liegt, desto größer ist ihr Einfluss auf die Nervenzelle. Gleichzeitig über verschiedene Synapsen ankommende Signale addieren sich in ihrer Wirkung. Übersteigt der aus den Botenstoffen aktivierte Reiz eine bestimmte Schwelle, aktiviert sich das Aktionspotential, wodurch das Neuron über einen Nervenzellfortsatz (Axon) und den dahinterliegenden Synapsen seinerseits Signale an verbundene Neuronen weiterleitet.

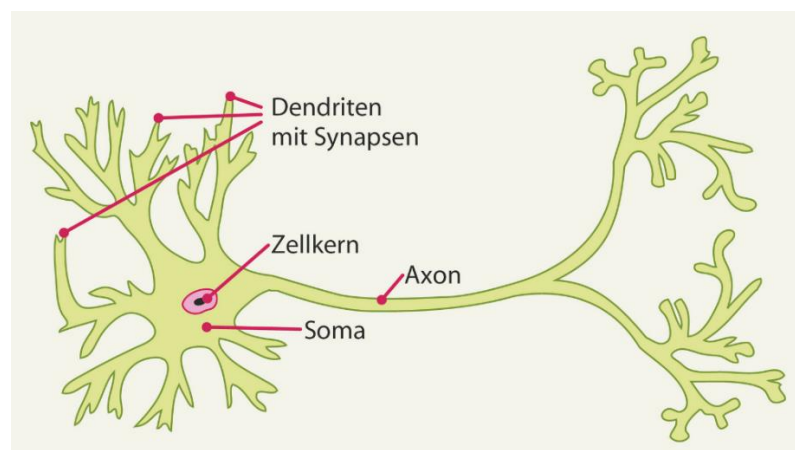


Abbildung 10 Neuron

Dabei sind weder die Verbindungen zwischen Neuronen noch die Bedingungen für die Signalübertragung (Reizschwelle) starr, es finden ständig Anpassungen anhand von Erfahrungen und Sinneseindrücken statt, die auf einen Mensch von Kindstagen an bis ins hohe Erwachsenenalter einwirken.

Zusätzlich übernehmen bestimmte Hirnareale spezifische Aufgaben, beispielsweise ist die primäre Sehrinde (V1) für das (visuellen) Erkennen zuständig, wobei sekundäre und tertiäre Bereiche (V2-V5) sich mehr und mehr auf einzelne Aspekte wie Form- oder Farbenerkennung spezialisieren (vgl. [Gro]).

Dies geht sogar so weit, dass einzelne Neuronenverbände das allgemeine Prinzip von Gesichtern verinnerlicht haben und weitere Verbände speziell auf individuelle Gesichter reagieren (s. [Geg]).

Ein Überblick zur historischen Entwicklung Neuronaler Netze findet sich im Anhang.

3.1.2 Vorteile

Aus der biologischen Beschreibung Neuronaler Netze lassen sich somit folgende Schlüsse ziehen, welche auch bei künstlichen Netzen gelten:

- Die Stärke des Systems liegt nicht darin begründet, dass die einzelne Recheneinheit (Neuron) schnell arbeitet, sondern dass das System aus vielen einfachen, massiv parallel arbeitenden Einheiten besteht.
- Neuronale Netze müssen nicht für eine bestimmte Aufgabe programmiert werden, sondern besitzen Lernfähigkeit, d.h. können sich selbstständig auf Basis von Trainingsbeispielen Wissen aneignen.
- Daraus resultiert auch die Fähigkeit zur Generalisierung: erfolgreiches Training ermöglicht Neuronalen Netzen Lösungen für ähnliche, untrainierte Probleme derselben Art zu finden.
- Eine direkte Konsequenz daraus ist auch eine hohe Fehlertoleranz gegenüber unklaren/verrauschem Eingangsmaterial.
- Auch sind Neuronale Netze i.d.R. äußerst stabil: Das System läuft auch dann noch, wenn einzelne Neuronen ausfallen.

Diese Eigenschaften Neuronaler Systeme stellen deutliche Vorteile im Vergleich zu traditionellen Experten-Entscheidungssystemen dar, welche meist nicht lernfähig sind, Entscheidungen auf Basis festgelegter Regeln fällen müssen und damit Schwierigkeiten bei unvorhergesehenen Problemen haben können.

3.2 Grundlagen

3.2.1 Künstliches Neuron

Ein künstliches Neuron ist im Grunde eine beschränkte, parametrisierte Funktion f von Eingaben x_i und Gewichten w_i mit $i = 1, \dots, n$ (vgl. [Wal]). f wird auch Aktivierungsfunktion genannt und bestimmt die Ausgabe y des Neurons.

Eingaben stammen dabei aus den Ausgaben anderer Neuronen oder Werten des beobachteten Prozesses (z.B. Graustufen-Wert eines Bild-Pixels). Gewichte regulieren den Einfluss eines Eingabe-Werts auf f und können dabei eine erregende ($w_i > 0$) oder hemmende ($w_i < 0$) Wirkung aufweisen. Bei $w_i = 0$ hat die Eingabe keinen Einfluss auf die Ausgabe des Neurons.

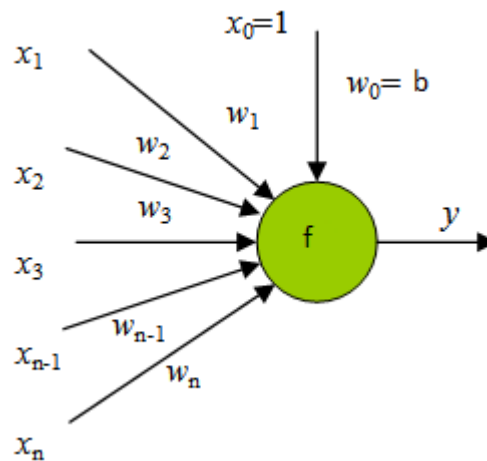


Abbildung 11 Künstliches Neuron

Als Eingangswert der Aktivierungsfunktion f (auch Netto-Input genannt) wird die gewichtete Summe der Eingabewerte ergänzt um dem Bias b gesetzt:

$$n = \sum_{i=0}^n (w_i \cdot x_i) = b + \sum_{i=1}^n (w_i \cdot x_i)$$

b ist hierbei das Gewicht des konstant auf 1 stehenden Eingangs x_0 und stellt den negierten Schwellwert (d.h. die Empfindlichkeit) des Neurons dar, womit dieser als Eingang im Neuron dargestellt werden kann.

Als Aktivierungsfunktion sind (u.a.) Schwellwertfunktionen oder sigmoide Funktionen denkbar:

Schwellwertfunktionen wechseln am Schwellwert $\theta = -b$ den Wert, sind ansonsten aber konstant, d.h.

$$f(n) = \begin{cases} y_0, & n < 0 \\ y_1, & n \geq 0 \end{cases}$$

f hat damit jedoch den Nachteil, dass die Funktion am Schwellwert nicht differenzierbar ist und ansonsten die Ableitung konstant 0 liefert, was das Lernen mit Backpropagation (s. Kapitel 3.4.2) zumindest erschwert (vgl. [StO-a]). Zudem ist die Funktion insofern instabil, dass eine leichte Variation am Eingang den Ausgang komplett kippen kann.

Besser eignen sich deshalb sigmoide Funktionen, welche stetig und an jeder Stelle differenzierbar sind. Häufig wird hierbei die logistische Funktion verwendet:

$$f(n) = \frac{1}{1 + e^{-n}}$$

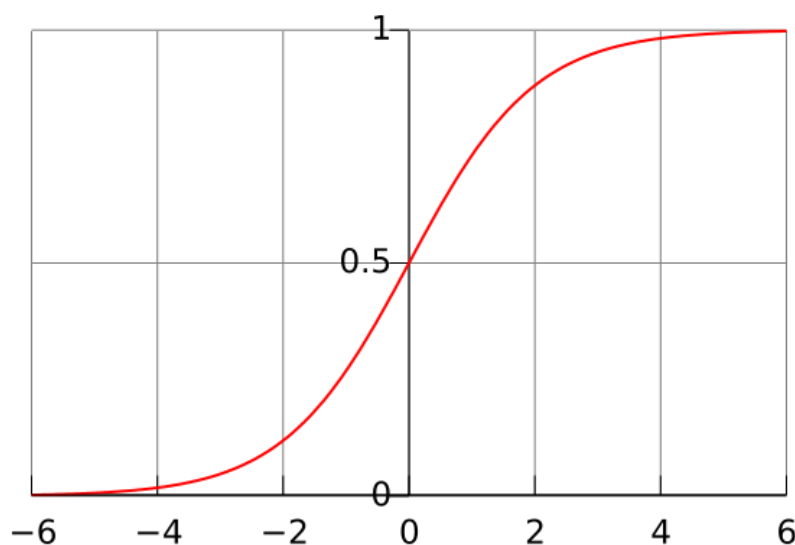


Abbildung 12 Logistische Funktion

Der Wertebereich der Funktion liegt damit in $(0,1)$. Zudem kann die (für das Lernverfahren benötigte) Ableitung der Funktion mit

$$f'(n) = \left(\frac{1}{1 + e^{-n}} \right)' = \frac{e^{-n}}{(1 + e^{-n})^2} = f(n) \cdot (1 - f(n))$$

angeschrieben werden. Für (sehr) große positive bzw. negative Werte von n kann die logistische Funktion auch als Näherung der Schwellwertfunktion gedeutet werden.

3.2.2 Künstliche Neuronale Netze

Der Zweck Künstlicher Neuronaler Netze (KNN) ist die Zuordnung eines Eingabewertes zu einem passenden Ausgangswert, d.h. die Klassifizierung des Eingangs. Meistens liegt dabei ein nichtlinearer Zusammenhang zwischen Ein- und Ausgabewerten vor.

Derart geforderte nichtlineare KNNs werden durch die Verknüpfung künstlicher Neuronen mit nichtlinearen Aktivierungsfunktionen abgebildet (vgl. [StO-b]), welche Schichtweise angeordnet sind.

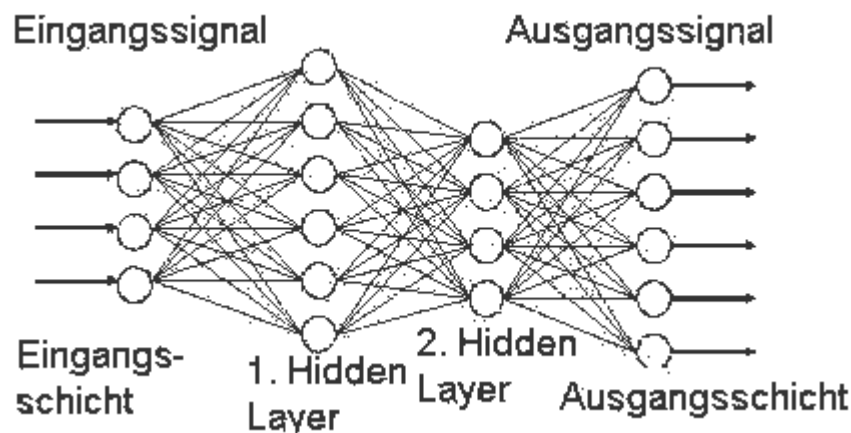


Abbildung 13 Künstliches Neuronales Netz

Dabei wird zwischen 3 Schicht-Typen unterschieden:

3.2.2.1 Eingangsschicht

Die Eingangsschicht (Input Layer) bzw. Eingabeschicht dient dazu, Roh- bzw. (vorbearbeitete) Eingangsdaten in das Netz einzuspielen. Die Anzahl der Neuronen in dieser Schicht orientiert sich normalerweise am Eingangssignal: handelt es sich bspw., wie in der vorliegenden Arbeit, um ein Bild in der Auflösung von 28x28 Pixel, werden 784 Neuronen definiert, d.h. pro Pixel ein Eingangswert.

3.2.2.2 Verdeckte Schicht(en)

Als nächstes kommen ein bis n verschiedene verdeckte Schichten (Hidden Layer) bzw. Zwischenlagen. Aufgabe dieser Schicht(en) ist insbesondere die Ausarbeitung simpler Muster und Strukturen, um aus diesen immer komplexere Merkmale extrahieren zu können. Das Wissen hierüber wird über den Trainings-Prozess angeeignet.

Dieser Ansatz ähnelt stark jenem des Gehirns, über mehrere Schichten verteilt z.B. visuelle Aufgaben wahrzunehmen.

Die Anzahl der verdeckten Schichten und Neuronen hängt stark vom Einsatzgebiet ab und lässt sich schwer allgemein angeben: eine zu geringe Zahl kann schnell die Leistungsfähigkeit des Netzes einschränken, eine zu hohe nicht nur die Trainingszeit massiv erhöhen, sondern auch zu Overfitting führen (s. Kapitel 3.5.4), gepaart mit einem Verlust der Generalisierungsfähigkeit. Experimente können, neben bekannten Heuristiken (vgl. [StE-a]), ein probates Mittel sein, um möglichst sparsame KNN zu finden, die dennoch genug Raum für Komplexität lassen.

Bei vielen verdeckten Schichten (eine genaue Anzahl ist nicht definiert) spricht man auch von „Deep Learning“.

3.2.2.3 Ausgangsschicht

Die Ausgangsschicht (Output Layer) bzw. Ausgabeschicht besteht aus so vielen Neuronen wie mögliche, zu unterscheidende Ergebnisse abgebildet werden müssen. Im Falle einer Schrifterkennung wären dies 62 Stück (Ziffern 0 bis 9 sowie die 26 Buchstaben des Alphabets in Groß- und Kleinschreibung).

Als Ergebnis einer Analyse von Eingangswerten eines KNN kann damit jenes Neuron der Ausgangsschicht gesehen werden, welches den höchsten Wert $f(n)$ aufweist.

3.3 Ausprägungen

Abhängig davon wie Neuronen bzw. die verschiedenen Schichten miteinander verknüpft sind, lassen sich unterschiedliche Ausprägungen von KNNs festmachen.

3.3.1 Feedforward-Netze

In vorwärtsgerichteten Netzwerken (Feedforward, FFN) fließt die Information stets in eine Richtung und zwar von der Eingangs- zur Ausgangsschicht (vgl. Abbildung 13). Die Netze sind somit zyklonfrei.

Im folgenden Kapitel 3.4 wird das Trainieren eines KNN beispielhaft anhand eines voll-verknüpften FFN gezeigt (bei diesem ist jedes Neuron einer Schicht mit allen Neuronen der Folgeschicht verknüpft), auch die Eigenentwicklung in Kapitel 5.3 bildet ein solches FFN nach.

3.3.2 Convolutional Neural Networks

Insbesondere in der Bildverarbeitung haben sich vorwärtsgerichtete Convolutional Neural Networks (CNN) bewährt (s. [Kar]). Dabei wird dem Umstand Rechnung getragen, dass bspw. die räumliche Struktur von Bildern nah beieinanderliegenden Pixeln eine höhere Beziehung einräumt als weiter entfernt liegende – eine Idee, die Anleihen am biologischen Vorbild des rezeptiven Feldes (vgl. [Wik-c]) nimmt.

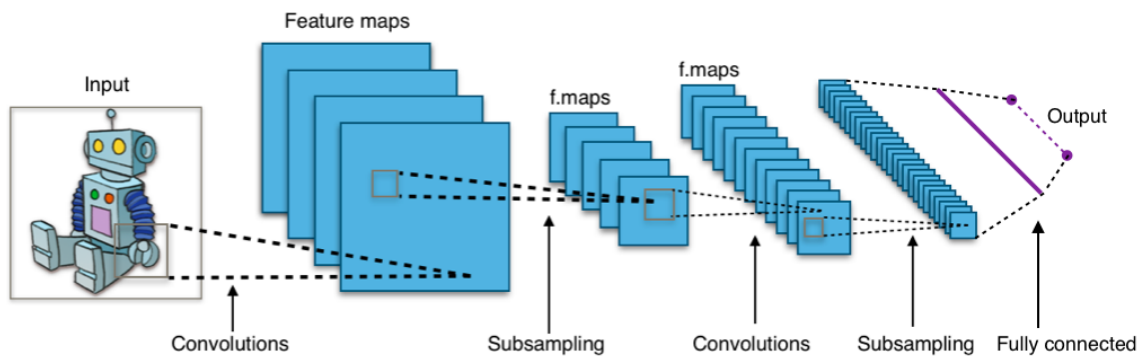


Abbildung 14 Convolutional Neural Network

Für den Aufbau von CNNs werden u.a. zwei besondere Schicht-Typen (wiederholt) eingesetzt (für die Klassifizierung am Ende werden normalerweise 2 voll-verknüpfte Schichten angeschlossen, wobei die Anzahl der Neuronen in der Ausgabeschicht den einzelnen, zu unterscheidenden Objekt-Klassen entspricht):

3.3.2.1 Convolutional Layer

Über die 2D-Anordnung des Eingangs wird schrittweise ein (vergleichsweise kleiner) Filterkern bewegt, wobei sich der Eingabe-Wert des entsprechenden Neurons im Convolutional Layer aus dem Skalarprodukt des Filterkerns (welcher als Gewichts-Matrix verstanden werden muss) und dem unterliegenden Bildausschnitt ergibt. Diese Überlagerung wird auch als (lokales) rezeptives Feld bezeichnet. Als Aktivierungsfunktion f wird bei CNNs üblicherweise Rectified Linear Unit (ReLU) gewählt:

$$f(x) = \max(0, x)$$

Nach vollständigem Durchlauf der Eingabe entsteht damit eine ebenfalls 2-dimensionale Feature Map.

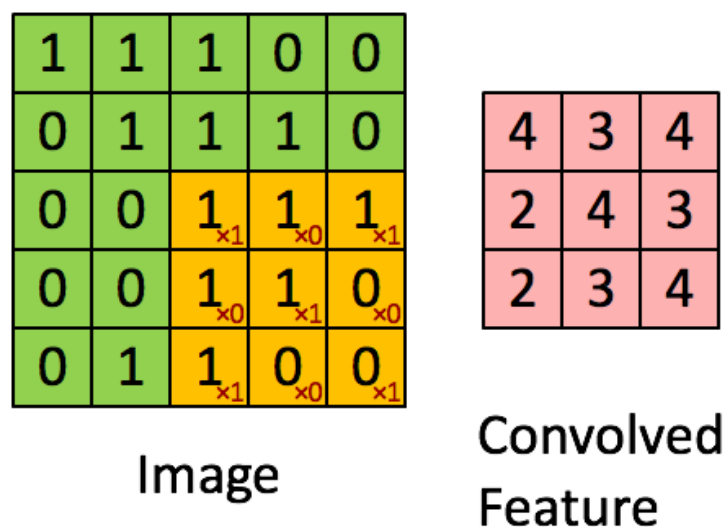


Abbildung 15 Feature-Map

Filterkerne (pro Eingabe ist die Definition mehrere möglich) können dabei als Muster oder Strukturen (d.h. einem Feature) in einem Bild interpretiert werden – die jeweils zugehörige Feature Map gibt somit an mit welcher Intensität ein solches Feature in einem lokalen (begrenzten) Bereich der Eingabe vorliegt bzw. reagieren benachbarte Felder in der Feature Map auf sich überlappende Bereiche der Eingabe.

3.3.2.2 Pooling Layer

Zweck des Poolings ist die Reduktion der Auflösung von Feature Maps. Üblicherweise wird dies per Max-Pooling erreicht: dabei werden die einzelnen Feature Maps des Convolutional Layers mit einem 2x2 Kernel schrittweise (mit Schrittlänge 2) durchlaufen und jeweils nur die Aktivität des aktivsten Neurons für die weitere Bearbeitung beibehalten.

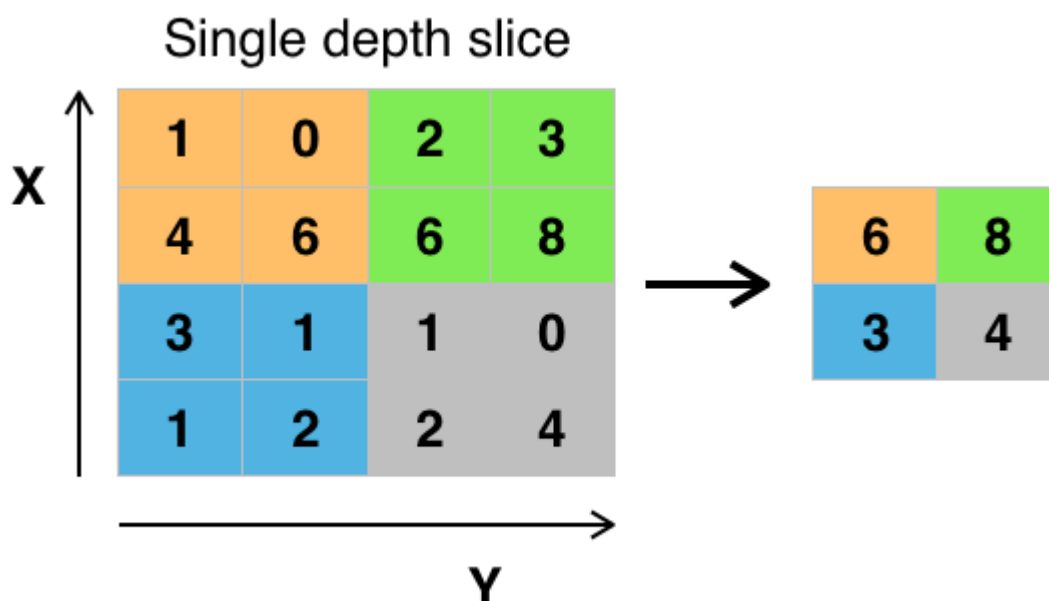


Abbildung 16 Max-Pooling mit Schrittweite 2

3.3.2.3 Vorteile

Eine Reihe von Gründen tragen dazu bei, dass CNNs zu den effizientesten KNNs gehören (vgl. [Cir]):

- Nachdem für jede Feature Map der gleiche Filterkernel und damit die gleichen (geteilten) Gewichte verwendet werden, müssen deutlich weniger Gewichte gespeichert und trainiert werden. Ebenso führt Pooling zu einem reduzierten Platzbedarf und erhöhter Berechnungsgeschwindigkeit. Als Konsequenz sind damit mehr Netz-Schichten möglich, womit sich wiederum komplexere Aufgaben lösen lassen können.
- Damit sind in weiterer Folge auch größere Eingabedaten, d.h. eine höhere Zahl an Inputneuronen, definierbar. Vorsicht gilt in diesem Fall aber vor dem „Fluch der

Dimensionalität“ (vgl. [StE-b]), worunter die steigende Anzahl erforderlicher Testdaten zum Erlernen des höherdimensionierten Raumes verstanden wird.

- Pooling ist ebenfalls eine gute Präventionsmaßnahme gegen Overfitting.
- Geteilte Gewichte haben sich als überaus robust gegenüber Translations-, Rotations-, Skalen- und Luminanzvarianz erwiesen (s. [LeC]).
- Zudem lassen sich CNNs effizient auf den immer leistungsfähigeren GPUs trainieren (s. [Kri]).

Die Entwicklung eines CNN mit Hilfe eines externen Frameworks wird in Kapitel 5.4 vorgestellt.

3.3.3 Rekurrente Neuronale Netze

Eine weitere Ausprägung Neuronaler Netze sind Rekurrente Neuronale Netze (RNN). Diese erlauben Verbindungen auch zwischen Neuronen der gleichen bzw. einer vorherigen Schicht oder mit sich selbst.

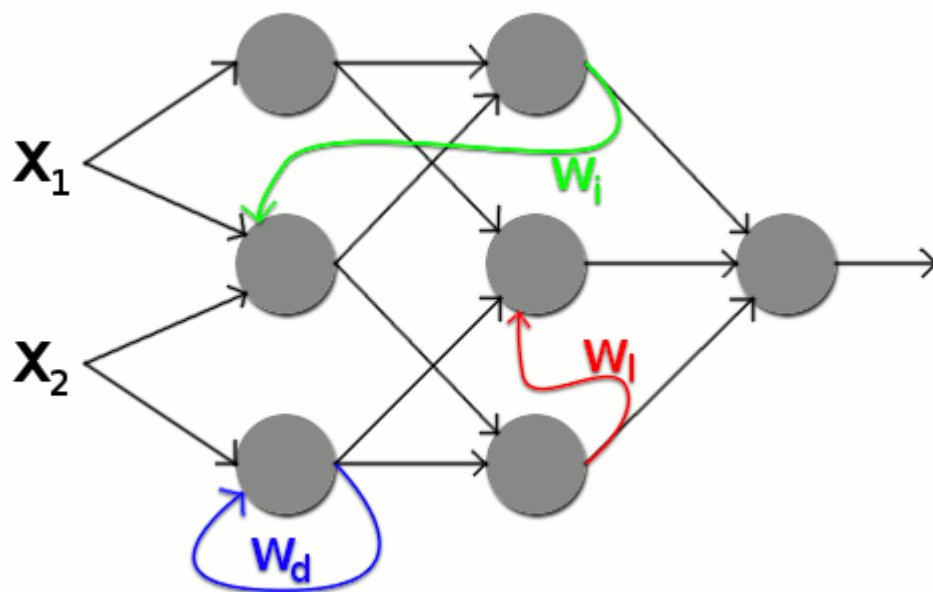


Abbildung 17 Rekurrentes Neuronales Netz

Diese Verbindungen können bspw. dazu verwendet werden um zeitlich codierte Informationen in den Daten zu entdecken. Praktisch bedeutsam sind RNNs v.a. dort, wo eine Verarbeitung von Sequenzen erforderlich ist (Spracherkennung, Interpretation von Videos, etc.).

RNNs stellen jedoch besondere Anforderung an ihr Training, auch sind (normale) RNNs nicht in der Lage, zu weit zurückliegende Informationen zu verknüpfen. Eine Verbesserung wurde hierbei mit der Entwicklung von Long short-term memory (LSTM) Netzen erreicht (s. [Hoc]).

RNNs bzw. LSTMs werden nicht weiter behandelt.

3.4 Trainieren

Das Wissen eines KNN liegt in seinen Gewichts-Werten²: von diesen Parametern hängt die (Klassifizierungs-)Leistung eines Netzes ab, weshalb Lernen zum Zwecke der Leistungs-Steigerung die Adaptierung eben jener Werte ist³.

Wie lassen sich aber passende Werte finden? Der Ansatz, diese so lange zu variieren bis jene Kombination gefunden wird, welche die höchste Zahl korrekter Treffer eines KNN liefert, scheitert nicht nur an der hohen Anzahl möglicher Variationen (manche KNN können Milliarden von Verbindungen aufweisen), sondern auch am Umstand, dass im allgemeinen kleinere Änderungen an den Gewichten die Anzahl der korrekten Treffer gar nicht ändert.

Als effizienter hat sich deshalb das überwachte Lernen (supervised learning) herausgestellt: Trainingsdaten, bestehend aus Paaren von Eingabe- und (erwarteter) Ausgabedaten, werden in ein KNN eingespeist, anschließend die Differenz aus dem tatsächlichen und dem korrekten Ausgabe bestimmt. Mithilfe dieser Differenz modifizieren Lernregel anschließend die Gewichte des Netzes.

3.4.1 Fehlerbestimmung

Wenngleich nicht die einzige Möglichkeit, wird häufig die quadratische Kostenfunktion zur Fehlerbestimmung herangezogen. Diese berechnet die mittlere quadratische Abweichung wie folgt:

$$E(w) = \sum_k (y_k - a_k)^2$$

Hierbei ist $E(w)$ die quadratische Kostenfunktion in Abhängigkeit aller Gewichte w des KNN, y_k der gewünschte und a_k der tatsächliche Output am Neuron k der Ausgangsschicht.

Ziel ist nun ein (idealerweise globales) Minimum der Fehlerfunktion per Näherungsverfahren zu finden, d.h. dazu passende Gewichts-Werte, was durch Abstieg in der Gradientenrichtung entlang der Fehlerfunktion erreicht wird (technisch gesehen definiert der Gradient

² Darunter fallen auch die Bias-Werte, welche jedoch (s. Kapitel 3.2.1) als Gewichte mit konstantem Eingang dargestellt werden können.

³ Hierbei handelt es sich um die am meisten verwendete Variante, alternative Strategien könnten auch die Modifikation der Aktivierungsfunktion f beinhalten.

den steilsten Anstieg, weshalb der negative Gradient für den steilsten Abstieg herangezogen wird. Mit Hilfe dieses Gradienten ist wiederum die Bestimmung der Gewichtsänderung möglich).

Bildlich lässt sich die Fehlerfunktion im 3-dimensionalen bei Reduktion der Parameter auf 2 Stück als Landschaft vorstellen, wo Senken (gesuchte) Minima darstellen:

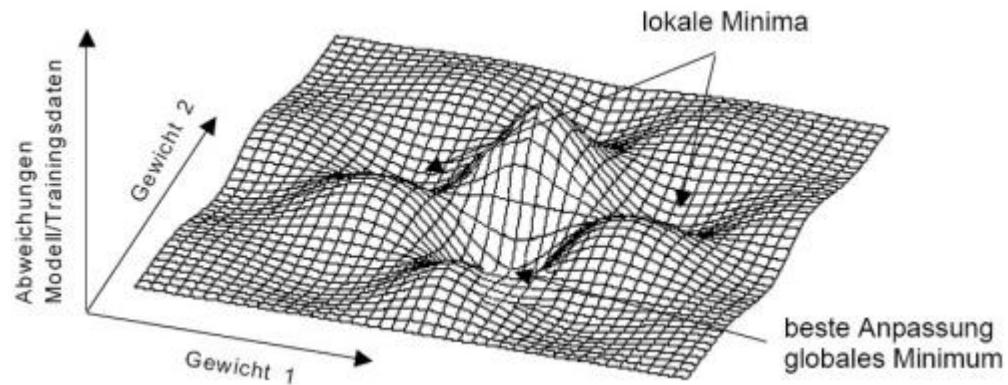


Abbildung 18 Fehlerfunktion mit 2 Gewichten

3.4.2 Backpropagation

Das Backpropagation-Verfahren ist eine Möglichkeit den Gradientenabstieg zu bestimmen und damit das KNN zu trainieren. Dabei wird der in der Ausgangsschicht errechnete Fehler (d.h. die Abweichung Ist-Soll) schrittweise und anteilig an die vorherigen Schichten zurückgespeist (backpropagiert), womit wiederum die Anpassung der einzelnen Gewichte möglich wird.

Basis des Verfahrens ist die Definition einer Gewichtsänderung Δw für nach folgender Vorgabe:

$$\Delta w \sim -\frac{\partial E}{\partial w}$$

Damit wird die konkrete Gewichts Anpassung bei Verwendung der quadratischen Kostenfunktion nach folgenden Formeln berechnet (für einen Beweis s. [Rei10], Seite 155 ff.):

3.4.2.1 Fehler in der Ausgangsschicht

Der Fehler δ_k für das Ausgabeneuron k ergibt sich aus

$$\delta_k = f'(n_k) \cdot (y_k - a_k)$$

Hierbei steht f für die Aktivierungsfunktion (bzw. f' für die Ableitung der Funktion), n_k für den Netto-Input des Ausgabeneurons k und y_k bzw. a_k für den gewünschten bzw. tatsächlichen Output.

Sollte die logistische Funktion für die Aktivierung verwendet werden, vereinfacht sich δ_k auf

$$\delta_k = f(n_k) \cdot (1 - f(n_k)) \cdot (y_k - a_k) = a_k \cdot (1 - a_k) \cdot (y_k - a_k)$$

Die Gewichtsänderung Δw_{jk} für die Verbindung zwischen dem Ausgabeneuron k und dem Neuron j der vorherigen, verdeckten Schicht ergibt sich mit

$$\Delta w_{jk} = \eta \cdot \delta_k \cdot a_j$$

a_j bezeichnet den (tatsächlichen) Ausgabewert des aus der vorherigen Schicht stammenden Neurons, η die Lernrate, welche definiert wie stark sich die Gewichte ändern sollen.

3.4.2.2 Fehler in den verdeckten Schichten

Für ein Neuron j einer verdeckten Schicht ergibt sich der (individuelle) Fehler δ_j aus

$$\delta_j = f'(n_j) \cdot \sum_k (\delta_k \cdot w_{jk})$$

Dabei ist n_j wiederum der Netto-Input des Neurons j und f' die Ableitung der Aktivierungsfunktion. Die Summenbildung erfolgt über alle mit j verknüpften Neuronen k aus der unmittelbaren Nachfolgeschicht.

Analog der Ausgabeschicht kann bei Verwendung der logistischen Funktion für f die verkürzte Schreibweise

$$\delta_j = a_j \cdot (1 - a_j) \cdot \sum_k (\delta_k \cdot w_{jk})$$

angeführt werden, auch die Anpassung des Gewichts zwischen einem Neuron i der Vorgängerschicht und dem Neuron j der (aktuellen) verdeckten Schicht ist ähnlich:

$$\Delta w_{ij} = \eta \cdot \delta_j \cdot a_i$$

3.4.3 Praxis

Die Anwendung des Backpropagation-Algorithmus umfasst somit folgende Schritte:

1. Initialisierung der Gewichte mit zufälligen Werten.
2. Einspeisung des (nächsten) Trainingsbeispiel x in das Netz.
3. Feed-Forward: Berechnung der Netz-Ausgabe.
4. Berechnung der δ und Δw in der Ausgabeschicht.
5. Backpropagation: Schichtweise Rückspeisung der Fehler aus der Ausgabeschicht, Berechnung von δ und Δw in den verdeckten Schichten.
6. Aktualisierung der Gewichte mit $w_{neu} = w_{alt} + \Delta w$.

7. Sprung retour auf Schritt 2.

Mögliche Abbruchbedingungen für diesen Zyklus sind:

- Alle Trainingsbeispiele wurden in das Netz eingespeist und die entsprechenden Fehler einkalkuliert.
- Eine bestimmte Anzahl von Trainings-Epochen wurde erreicht. Unter einer Trainings-Epoche versteht man den vollständigen Durchlauf aller Trainingsbeispiele.
- Die Fehlerrate (welche bspw. zwischen Schritt 3 und Schritt 4, nach Durchlauf von N Trainingsbeispielen, berechnet werden könnte) sinkt unter eine zuvor definierte Schwelle (Early stopping).

3.4.3.1 Stochastisches Gradientenverfahren

Bei der Zyklus-Beschreibung wurde unter Schritt 6 festgehalten, dass die Gewichtsänderungen mit jedem Trainingsbeispiel geschrieben werden. Eine solche Strategie wird auch Stochastisches Gradientenverfahren genannt und hat im Gegensatz zu dem als Batchlearning bekannten Ansatz, zuerst alle Trainingsbeispiele in das Netz einzuspeisen und erst anschließend die Summe der berechneten Gewichtsänderungen zum jeweiligen Gewicht zu addieren (d.h. pro Trainingsbeispiel Schritt 6 zu überspringen), den Vorteil einer höheren Ausführungsgeschwindigkeit.

Nachteilig kann sich jedoch auswirken, dass die (globale) Fehlerminimierung nicht gleichmäßig konvergiert. Als Kompromiss hat sich deshalb die Verwendung von Mini-Batches etabliert: dabei werden die errechneten Gewichtsänderungen aus n zufällig gewählten Trainingsbeispielen kumuliert und für die Adaptierung der Gewichte herangezogen.

Die zufällige Wahl von n Beispielen sollte dabei (bestmöglich) sicherstellen, dass die durchschnittlichen Kosten (bezogen auf die Fehlerfunktion) der Mini-Batch-Auswahl in etwa den durchschnittlichen Kosten aller Trainingsbeispiele entsprechen.

Ein guter Wert für n könnte dabei bei 32 liegen (vgl. [Ben12]).

3.5 Gängige Probleme (und mögliche Lösungen)

Neben den bereits tlw. bei den einzelnen, bisherigen Punkten genannten Problemen, stößt man bei der Implementierung von KNN regelmäßig auf weitere. Diese sollten im Folgenden zumindest überblicksartig besprochen werden.

Oft betreffen Probleme die Hyperparameter eines KNN: dies sind zu einem großen Teil jene Netz-Eigenschaften, welche nicht durch das Trainieren bestimmt werden sondern, ganz im Gegenteil, das Trainings-Modell (mit)definieren und somit a-priori-Wissen (bzw. entsprechende Erfahrung) voraussetzen.

Vorausgeschickt sei aber auch, dass es keinen Königsweg zum Finden einer optimalen Lösung (d.h. eines globalen Minimums von $E(w)$) gibt – aber auch suboptimale Lösungen sind besser als irgendwelche Lösungen.

3.5.1 Initialisierungs-Werte

Bei der Initialisierung der Gewichts-Werte sollten mehrere Faktoren berücksichtigt werden:

- Werden alle Gewichts-Werte mit 0 belegt, ist Lernen gar nicht möglich, da nach Definition von Δw der Gradient (und somit die Gewichtsänderung) 0 wäre (vgl. [StE-c]). Generell führt die Initialisierung mit identen Gewichts-Werten zum Problem, dass sich alle Neuronen der inneren Schicht(en) gleich verhalten und somit ein unflexibles, praktisch unbrauchbares KNN vorliegt.
- Werden zu hohe Startwerte festgelegt, droht bei Verwendung sigmoider Aktivierungsfunktionen eine Sättigung, d.h. die Ableitung der Funktion und damit auch der Gradient ist (nahezu) 0. Dies würde eine (unnötigen) Verlangsamung des Lernprozesses bedeuten.
- Zu niedrige Startwerte können hingegen den Verlust der benötigten Nicht-Linearität des KNN bedeuten. Zudem könnte ein Eingangssignal auf eine verschwindend geringe Größe reduziert werden.

Ideal ist somit die Initialisierung mit Zufallswerten, wie sie bspw. die Gauß-Verteilung liefert. Das 2010 vorgestellte Xavier-Initialisierungs-Verfahren zur Bestimmung der Verteilungs-Varianz verhindert zudem die unkontrollierte Streuung des Eingangssignals auf einen sehr niedrigen oder sehr hohen Wert (vgl. [Jon]).

3.5.2 Lernrate

Die Wahl einer passenden Lernrate η ist ebenfalls eine Kunst an sich und lässt sich allgemein nicht angeben: Ein geringer Wert kann zu einem langen Lernprozess führen, da beispielsweise flache Plateaus der Fehlerfunktion nur langsam durchlaufen werden (a).

Ein hoher Wert hingegen führt zu großen Gewichtsveränderungen, wodurch die (globale) Minimierung von $E(w)$ leicht verfehlt werden kann (b). Auch droht die Gefahr einer Oszillation (c): der plötzliche Wechsel von einem Gradienten auf einen zweiten Gradienten gleichen Betrags aber mit unterschiedlichem Vorzeichen lässt den gesamten Fehler-Optimierungsprozess stagnieren.

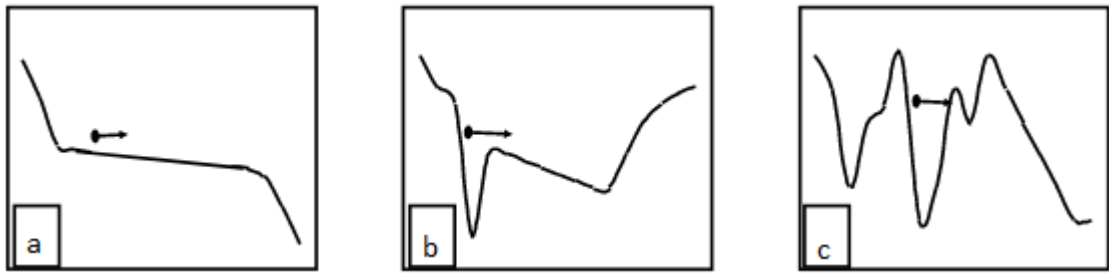


Abbildung 19 Mögliche Lernverläufe bei Optimierung von $E(w)$

Eine mögliche Strategie könnte deshalb lauten, anfangs ein grobes Lernen mit einer höheren Lernrate anzustreben und bei fortschreitendem Training die Rate (mehrfach) zu verringern.

3.5.3 Lokale Minima

Das Finden eines Minimums der Fehlerfunktion gleicht dem Rollen einer Kugel in einer Schlucht: idealerweise rollt die Kugel bis in die (globale) Senke (a).

In der Praxis treten jedoch oft lokale Minima auf (die Fehlerfunktion ist im Allgemeinen nicht konvex), welche das Erreichen eines globalen Minimums verhindern. Der Analogie folgend, steckt die Kugel in einer solchen Mulde fest (b).

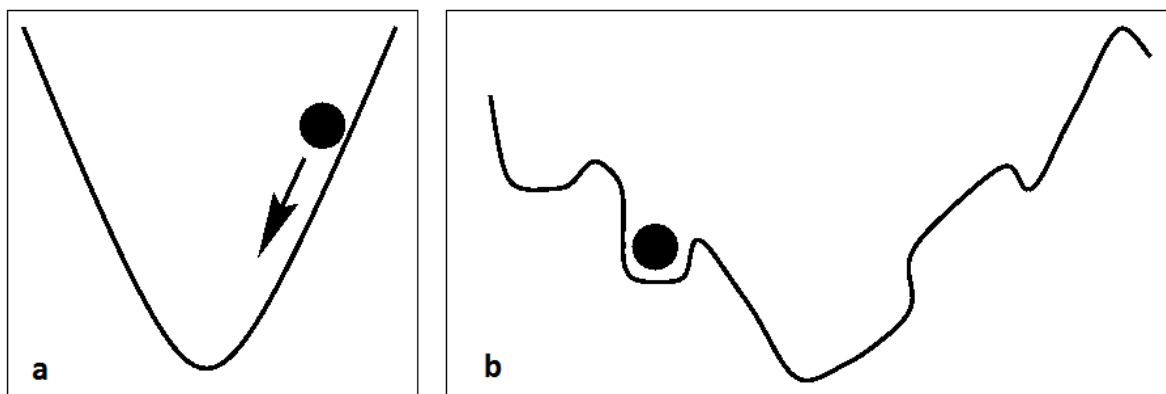


Abbildung 20 Lokale Minima

Eine Momentum genannte Technik entschärft solche Situation, indem nicht nur der aktuelle Gradient zum Zeitpunkt t zur Bestimmung der Gewichtsänderung herangezogen wird, sondern auch (ein Teil) der vorherigen (zum Zeitpunkt $t - 1$ erfolgten) Gewichts-Anpassung:

$$\Delta w(t) = \eta \cdot \delta \cdot a + \mu \cdot \Delta w(t - 1)$$

Somit wird eine „Trägheit“ mit einbezogen, was dafür sorgen soll, dass lokale Senken bzw. flache Plateaus der Fehlerfunktion einfach „überrollt“ werden und damit die Fehlerfunktion schneller konvergiert.

Der Wert von μ liegt zwischen 0 und 1, mit Bedacht sollten jedoch gleichzeitig hohe Werte für μ und (der Lernrate) η wählen, da die Schrittweite in einem solchen Fall schnell (zu) hohe Werte annehmen und damit auch ein globales Minimum übersprungen werden könnte.

3.5.4 Überanpassung

Lernen erfolgt über das Einspielen von Trainingsbeispielen und darauffolgende Extraktion von allgemeinen Konzepten um auch unbekannte (nicht trainierte) Beispiele klassifizieren zu können.

Im Falle einer Überanpassung (Overfitting) lernt das KNN jedoch nur Trainingsdaten auswendig und versagt somit bei der Klassifizierung anderer Daten. Dies lässt sich auch grafisch darstellen:

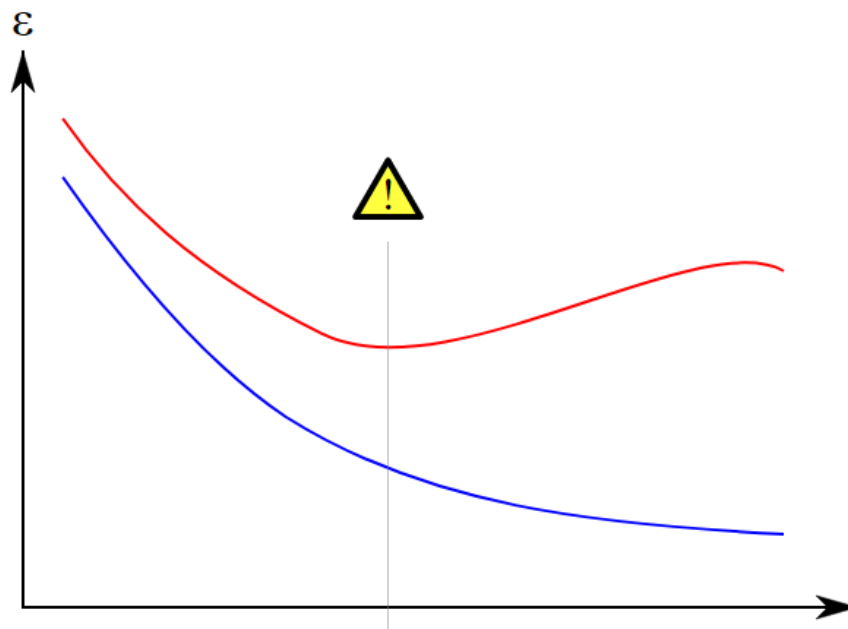


Abbildung 21 Überanpassung

Während die Fehlerrate bei Klassifizierung von Trainingsdaten (blau) mit fortschreitendem Lernprozess sinkt, bleibt die Fehlerrate der Testdaten (rot) immer (deutlich) darüber. Trainings- und Testdaten sind hierbei disjunkte (Teil-)Mengen aus dem gleichen Set von im Vorhinein klassifizierten, möglichst repräsentativen Daten.

Ursachen können vielfältiger Natur sein: neben einer geringen Varianz der Trainingsdaten auch eine ungünstige Wahl der KNN-Parameter (Anzahl der verdeckten Neuronen).

Eine einfache Strategie gegen Überanpassung sind somit zusätzliche Trainingsdaten: für ein sinnvolles Training sollte die Anzahl der Trainingsdaten mindestens um den Faktor 10 höher sein als die Anzahl der freien Parameter (Gewichte) im KNN (s. [Hay99], Seite 230).

Stehen nicht genügend Trainingsdaten zur Verfügung, könnten diese auch künstlich erzeugt werden, bspw. durch (leichtes) Rotieren oder Strecken von bestehenden Testbildern.

Insbesondere wenn Trainingsdaten und die Struktur des KNN fixiert sind, ist mit der Regulierung (Regularization) eine weitere Technik zur Vermeidung von Überanpassung verfügbar: dahinter steckt die Idee, dass ein zusätzlicher Term in der (zu minimierenden) Kostenfunktion dafür sorgt, dass hohe Gewichtswerte verhindert werden, womit KNNs angehalten wären weniger auf Ausreißer in den Testdaten zu reagieren und stattdessen mehr zu Generalisieren, somit die Wahrscheinlichkeit einer Überanpassung sinkt.

Die gängige L2-Regulization wird dabei wie folgt definiert und einfach zu $E(w)$ addiert:

$$L2(w, n) = \frac{\lambda}{n} \sum_w w^2$$

Die Summe wird damit über das Quadrat aller Gewichte w des KNN gebildet, n ist die Trainings-Batch-Größe und λ der Regulierungs-Parameter. Die Wahl von λ ist, wie so oft, grundsätzlich problemabhängig, bei einem großen Wert Schrumpfen die Gewichte jedoch gegen 0. Oft wird deshalb das Intervall $[0,001; 0,1]$ herangezogen.

Eine weitere, sich als überaus effizient herausgestellte und dennoch simple Strategie ist Dropout (s. [Sri14]): während der Trainingsphase wird jedem Neuron eine Wahrscheinlichkeit p zugeteilt, mit der die Neuronen-Ausgabe im kommenden Berechnungsschritt auf 0 festgelegt wird. Falls das KNN auf dieses Neuron für die Klassifizierung eines (auswendig gelernten) Datensatzes angewiesen war, muss nun doch auf andere Neuronen, die möglicherweise allgemeinere Strukturen abbilden, zugegriffen werden.

3.6 Universal Approximation Theorem

Die Bedeutung (und Stärke) Neuronaler Netze stammt auch daher, dass sich mit ihrer Hilfe beliebige nicht-konstante Funktionen f beliebig genau konstruieren lassen, d.h. dass die Ausgabe eines Neuronalen Netzes bei Eingabe jedes beliebigen Wertes x beliebig nahe beim Funktionswert $f(x)$ liegt. Nachdem sich (fast) jedes Problem als Funktion darstellen lässt, ist die Implikation immens.

Diese Eigenschaft beruht auf dem Universal Approximation Theorem: dieses besagt, dass ein FFN mit einer einzelnen verdeckten Schicht (mit endlich vielen Neuronen) stetige Funktionen auf kompakten Submengen von \mathbb{R}^n approximieren kann.

Das Theorem trifft jedoch keine Aussage darüber, wie viele Neuronen tatsächlich für eine Approximation erforderlich sind, zudem wird nicht garantiert, dass Trainings-Algorithmen auch die passenden Parameter finden bzw. keine Überanpassung passiert.

Forschungen haben jedoch gezeigt, dass in „Deep Learning“-Netzwerken, d.h. bei Verwendung von mehr verdeckten Schichten, die Näherung mit steigender Tiefe besser wird und auch die Generalisierungs-Fähigkeit weniger stark gefährdet ist (s. [Quo]).

4 Präzisierung

Im folgenden Kapitel werden ausgehend von den bisherigen theoretischen Betrachtungen (Nicht-)Ziele der Arbeit definiert sowie weiteres Hintergrundwissen für die Implementierung gegeben.

4.1 Ziele (und Nicht-Ziele)

Die Arbeit soll primär einen Überblick auf die behandelten Themen liefern. Ausgehend von den Analysen in Kapitel 2 und 3 ergeben sich somit folgende Ziele:

- Vorstellung grundlegender (und für die Arbeit relevanter) Bildvorverarbeitungsschritte.
- Die Ausarbeitung einer kompakten Übersicht auf fundamentale Konzepte von Neuronalen Netzen.
- Daran aufbauend die Entwicklung einer Software, welche eine handschriftlich erfasste IBAN erkennen und verarbeiten kann. Diese Software besteht im Gesamten aus 3 Teilen:
 - Die Extraktion einzelner Zeichen aus einer handschriftlich erfassten und anschließend digitalisierten IBAN.
 - Eine Zeichenerkennung mit Hilfe eines zuvor trainierten Neuronalen Netzes.
 - Die Interpretation (und Validierung) der erkannten Zeichen als österreichische (bzw. einer struktur-ähnlichen) IBAN .
- Die Implementierung ist flexibel genug, um die Auswirkung verschiedener Parameter erproben zu können.
- Die Entwicklung eines Neuronalen Netzes wird zum Zwecke eines direkten Vergleichs sowohl in einer kompletten Eigenimplementierung (basierend auf den vorgestellten, grundlegenden Konzepten) als auch auf Basis eines entsprechenden (professionellen) Frameworks durchgeführt.

Kein Ziel der Arbeit ist hingegen:

- Eine allumfassende Darstellung, welche jede Eventualität der Thematik abdeckt.
- Der Fokus auf die Verarbeitung der Handschrift in allen möglichen (Aufnahme-)Situationen.
- Der Fokus auf die Entwicklung eines (maximal) effizienten und hochoptimierten Codes.

Daraus folgt:

- Für die Zeichen-Extraktion werden Bedingungen festgelegt, die bei der handschriftlichen Erfassung der IBAN gelten müssen.
- Bei Umsetzung der Software wird kein Fokus auf Bedienerfreundlichkeit gelegt, weshalb auch keine eigene GUI (auch nicht in Form einer App) programmiert wird.
- Für die Software ist keine konkrete Implementierung in bestehenden Code bzw. ein unmittelbarer Praxiseinsatz geplant. Die Software ist mehr Vehikel für entsprechende Analysen.
- Performance-Messungen werden, bedingt durch die vielen mögliche Parameter und lange Messzeiten, nur rudimentär erhoben.

4.2 IBAN

Als praktischer Anwendungsfall eines auf Handschrift-Erkennung trainierten Neuronalen Netzwerkes wird das Lesen einer IBAN demonstriert. Dies hat nicht nur den Vorteil, dass ein begrenzter Zeichensatz (Buchstaben und Ziffern) verwendet wird, auch kann eine Validierung der erkannten Zeichen syntaktisch erfolgen, da die Struktur einer IBAN klar definiert ist.

4.2.1 Hintergrund

Eine IBAN (International Bank Account Number) ist eine standardisierte, internationale Beschreibung einer Bankkontonummer. Das in der Norm ISO 13616-1:2007 definierte Verfahren dient vor allem der Vereinheitlichung im internationalen Zahlungsverkehr, was nicht nur Banken sondern auch Privatpersonen zugutekommt.

4.2.2 Aufbau

Der grundsätzliche Aufbau einer IBAN gliedert sich in einen 2-stelligen Ländercode (Großbuchstaben, definiert in ISO 3166-1), einer 2-stelligen Prüfsumme (gemäß ISO 7064 bestehend aus Ziffern) und maximal 30 Stellen für die Kontoidentifikation, auch BBAN (Basic Bank Account Number) bezeichnet:

Somit kann eine IBAN aus maximal 34 Stellen bestehen, ist in der Regel (länderabhängig) jedoch deutlich kürzer.

4.2.2.1 Österreich

In Österreich besteht eine IBAN aus 20 Stellen: Nach dem Ländercode „AT“ und der 2-stelligen Prüfsumme folgt die 5-stellige, pro Bank-Institut eindeutige Bankleitzahl, anschließend die Kontonummer, jeweils bestehend aus Ziffern.

Österreichische IBAN (20 Stellen)																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
AT		Prüfs.		Bankleitzahl						Kontonummer									

Abbildung 22 Aufbau einer österreichischen IBAN

Ist die Kontonummer weniger als die maximal möglichen 11 Stellen lang, wird diese mit führenden Nullen direkt nach der Bankleitzahl aufgefüllt.

Diese grundsätzliche Struktur (nur die ersten beiden Stellen sind Buchstaben, die restlichen Ziffern) besteht bspw. auch bei einer deutschen oder schweizerischen IBAN, gilt jedoch nicht für jedes Land, welches am IBAN-System teilnimmt.

4.2.3 Darstellung

Aus Gründen der Lesbarkeit (u.a. für die Darstellung auf Kontoauszügen und Rechnungen oder bei Eingabe in elektronischen Formularen) wird die IBAN oft in Vierergruppen, mit Leerzeichen getrennt, dargestellt:

AT02 2050 3021 0102 3600

4.2.4 Validierung

Die Validierung einer IBAN basiert auf dem Standard ISO 7064 mod 97-10 und erfolgt nach folgendem Muster:

1. Zuerst wird die IBAN neu geordnet, indem die Reihung BBAN, codierter Ländercode und Prüfsumme vorgenommen wird. Die Codierung des Ländercodes erfolgt durch Position des Buchstabens im lateinischen Alphabet + 9 (aus A wird 10, aus B 11, ..., aus Z 35). Somit generiert man aus der IBAN eine Zahl.
2. Diese Zahl wird ganzzahlig durch 97 geteilt, der Rest aus der Division bestimmt (Modulo 97).
3. Bei einem Rest 1 ist die IBAN korrekt, ansonsten falsch.

4.2.5 Beispiel

Aus der Bankleitzahl 20403 und der Kontonummer 2101023600 wird von einer österreichischen Bank die IBAN AT022050302101023600 berechnet.

Zur Validierung wird die IBAN zuerst neu sortiert:

2050302101023600AT02

Anschließend erfolgt die Codierung des Länderkürzels:

2050302101023600102902

Der Rest der Operation aus mod 97 ist 1, die IBAN ist somit valide.

4.3 Trainingsdaten

Da eine IBAN wie beschrieben aus Buchstaben und Ziffern besteht, muss das Neuronale Netz auf die Erkennung von (zumindest) 62 (handschriftlichen) Zeichen trainiert werden:

- 26 Kleinbuchstaben, a-z
- 26 Großbuchstaben, A-Z
- 10 Ziffern, 0-9

Hierfür ist eine entsprechend hohe Zahl von Trainingsdaten notwendig, welche u.a. die amerikanische Standardisierungsbehörde NIST (National Institute of Standards and Technology) zur freien Verfügung stellt.

4.3.1 MNIST-Datenbank

Geradezu als Klassiker gilt der MNIST-Datensatz (Modified National Institute of Standards and Technology, s. [MNI]), eine Sammlung von 70.000 Einzel-Bilder handgeschriebener Ziffern, aufgeteilt in 60.000 Trainings- und 10.000 Test-Bilder.



Abbildung 23 MNIST-Auszug

Die Daten sind eine Bearbeitung von NIST SD-1 (Special Database 1) und SD-3. Hintergrund für die Adaptierung ist der Umstand, dass die beiden Original-Datensätze der NIST von 2 verschiedenen Gruppen (SD-3 von Angestellten des United States Census Bureau, SD-1 von High-School-Schülern) her stammen, wobei SD-3 als Trainings- und SD-1 als Test-Daten gedacht waren. Diese Diskrepanz hat sich jedoch als problematisch beim Maschinellen Lernen herausgestellt.

Als Konsequenz wurde von Yann LeCun et al. eine Neugruppierung vorgenommen: jeweils die Hälfte der Trainings-Daten sowie Test-Daten stammen nun von beiden original NIST-Datensätzen, auch wurde darauf geachtet, dass kein Schreiber gleichzeitig in den Trainings- und Test-Daten vorkommt.

Im Zuge dessen wurde neben einer einheitlichen Skalierung der Bilder auf eine Auflösung von 28x28 Pixel auch zeitgleich ein Antialiasing der Bilder durchgeführt, was Graustufen-Werte verfügbar machten.

4.3.2 NIST SD-19

NIST SD-19 (s. [NIS]) ist eine Erweiterung dieser Sammlung um Klein- und Groß-Buchstaben auf insgesamt 62 Zeichen, verteilt über 814.255 Einzelbilder in der einheitlichen Auflösung von 128x128 Pixel. Als Datengrundlage dienen 3.669 handschriftlich ausgefüllte Formularbögen, von welchen die einzelnen Buchstaben bzw. Ziffern extrahiert wurden.

Bei der Erstellung der Formulare wurde darauf geachtet, dass von jedem Probanden der gesamte Zeichenvorrat angeführt werden musste, wobei viele Zeichen auch mehrfach abgefragt wurden. Die Daten stammen wiederum von Angestellten des United States Census Bureau sowie High-School-Schülern.

HANDWRITING SAMPLE FORM

NAME [REDACTED] DATE 8-3-89 CITY MINDEN CITY STATE MI ZIP 48456

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

0123456789 0123456789 0123456789

87 701 3752 80759 960941

87 701 3752 80759 960941

158 4586 32123 832656 82

158 4586 32123 832656 82

7481 80539 419219 67 904

7481 80539 419219 67 904

61738 729658 75 390 5716

61738 729658 75 390 5716

109334 40 625 4234 46002

109334 40 625 4234 46002

gyxlakpdsbtzirumwffqjenhocv

gyxlakpdsbtzirumwffqjenhocv

ZXSBNGECHMYWQTKFLUOHPIRVDA

ZXSBNGECHMYWQTKFLUOHPIRVDA

Please print the following text in the box below:

We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

Abbildung 24 NIST SD-19 Beispiel-Formular

Die extrahierten Daten können in verschiedenen Strukturen abgefragt werden, u.a. gruppiert nach Formular-Reihe, Autor, Formular-Feld oder Zeichen-Klasse (ASCII-Code im Hexadezimal-Format). Jede Segmentierung wurde zusätzlich manuell überprüft, weshalb die verbleibende Rate an falscher Zuordnung auf ca. 0,1% geschätzt werden kann.

Mit Version 2 des Datensatzes, verfügbar seit 2016, liegen die Einzelbilder auch im PNG-Format vor, was die weitere Verwendung durchaus vereinfacht.

5 Implementierung

Die theoretischen Überlegungen aus Kapitel 2 und 3, ergänzt um die relevanten Punkte aus Kapitel 4, bilden im Folgenden die Grundlage für die Implementierung einer eigenen, einfachen Zeichenextraktion und eines Neuronalen Netzwerkes. Für letzteres werden 2 Alternativen gezeigt: eine komplette Selbstentwicklung (unter Beachtung grundlegender Konzepte und beschränkt auf den MNIST-Datensatz) sowie die Verwendung eines speziellen Deep-Learning-Frameworks.

5.1 Grundlagen

5.1.1 Programmablauf

Die fertige Implementierung umfasst folgende, getrennt aufrufbare Schritte

1. Training des KNN auf Basis entsprechender Trainingsdaten aus (M)NIST-Beständen.
2. Vorverarbeitung eines handschriftlich erfassten Textes zur Extraktion einzelner Zeichen.
3. Klassifizierung der extrahierten Zeichen mit Hilfe des KNN.
4. Inhaltliche Prüfung der erkannten Zeichen auf Übereinstimmung mit einer IBAN.



Abbildung 25 Angesetzter Programm-Ablauf

5.1.2 Entwicklungswerkzeuge

Die Umsetzung erfolgt in Java (s. [Jav]), Version 1.8.0_131 (64-Bit), unter Zuhilfenahme der Entwicklungsumgebung Eclipse (s. [Ecl]), Version 4.6.3. Entwickelt und getestet wird unter Windows 10 (64-Bit) auf einem Notebook mit Intel-CPU (i5-6300U) und integrierter Grafikkarte (Intel HD 520).

Die Entscheidung für Java fiel deshalb, da es sich um die primäre Programmiersprache im beruflichen Umfeld handelt.

Zusätzlich werden je Umsetzungsschritt weitere Bibliotheken und (externe) Tools eingesetzt, eine detailliertere Beschreibung erfolgt direkt bei den einzelnen Punkten.

5.1.3 Vorbearbeitung der Trainingsbilder

5.1.3.1 *MNIST*

Der auf der offiziellen Homepage erhältliche MNIST-Datensatz liegt in einem speziellen Binärformat (vgl. [Idx]) vor, was die direkte Bearbeitung erschwert und deshalb eine vorherige, einmalige Konvertierung der einzelnen Bilder in das PNG-Format, zwecks Verwendung in der Eigenimplementierung des KNN, nötig macht.

Die entsprechende Konvertierung erfolgt in der (statischen) Methode `NNPreparations.convertidx` (Projekt `nnetwork`).

5.1.3.2 *NIST SD-19*

Wie in Kapitel 4.3.2 erwähnt, stehen die Bilder aus NIST SD-19 grundsätzlich in der Auflösung 128x128 zur Verfügung. Um hier Kompatibilität mit dem MNIST-Datensatz zu schaffen (u.a. um die Anzahl der Eingangsneuronen des KNN nicht zu erhöhen), werden diese deshalb in einem ersten Schritt in das Format 28x28 Pixel konvertiert.

Hierzu wird auf ein entsprechendes Github-Projekt zurückgegriffen (s. [Git-a]), welches die Konvertierung der einzelnen Bilder nach folgendem, auch beim MNIST-Datensatz eingesetztem Muster durchführt (Klasse `MnistSD19Preprocess`, Projekt `d14j`):

- Bestimmung des Vorder- und Hintergrunds im Original-Bild unter Verwendung eines entsprechenden Schwellwertes.
- Berechnung des Vordergrund-(Massen-)Schwerpunkts und Verwendung desselbigen zur Wahl des kleinstmöglichen Rechtecks rund um den Vordergrund.
- Skalierung dieser Auswahl auf 20x20 Pixel.
- Setzen der so generierten Abbildung in ein neues Bild der Auflösung 28x28 Pixel am Punkt (4,4), womit sich ein 4-Pixel breiter Rand ergibt.

Als Original-Datenquelle wird jenes nach dem Bildwert-klassifizierte Archiv gewählt, d.h. die entsprechenden Bilder liegen sortiert nach Zeichen, mit dem Zeichen-ASCII-Hex-Code als Ordnername, vor. Zusätzlich ist für jedes Zeichen die Unterteilung nach Formularreihe (hsf_*) und Trainingsdaten (train_*) gegeben.

Mit der Konvertierung werden nicht nur alle Testdaten aus hsf_* in der gleichen Ordner-Hierarchie zusammengefasst, sondern auch, durch leichte Adaptierung des Original-Codes, der Ordnername auf den entsprechenden ASCII-Dezimalwert festgelegt (was v.a. der späteren Verwendung bei der Umsetzung des KNN mit DL4J zugutekommt, s. Kapitel 5.4.2.3).

Die Konvertierung erfolgt getrennt für Test- und Trainingsdaten, womit auch 2 neue, entsprechend befüllte Ordner generiert werden.

5.1.4 IBAN-Prüfung

Die Validierung einer IBAN gelingt, Kapitel 4.2.4 folgend, durch Transformation des IBAN-Strings in eine Zahl und anschließender Berechnung von Modulo 97.

Da es sich hierbei um einen standardisierten Prozess handelt, wird aus praktischen Gründen die externe Bibliothek `java-iban` (s. [Git-b]) verwendet.

Eine Einrichtungs-Anleitung findet sich im Anhang.

5.2 Bildvorverarbeitung

5.2.1 Anforderung

Entgegen den ursprünglichen Erwartungen und Absichten, haben erste Gehversuche die Komplexität der Extraktion von Zeichen unter (auch typischen) Alltagssituation gezeigt. Zeichen-Überlappungen, Rotationen und Streckungen, Szenen-Beleuchtung, variable Strichstärke u.v.m. sind alles andere als trivial und hätten deshalb durchaus Potential für ein eigenständiges Diplomarbets-Thema.

Um dennoch eine gewisse Dynamik (Einspeisen eigener Handschrift) bereitzustellen und damit die grundlegende Verfahren aus Kapitel 2 präsentieren zu können, werden folgende Vorgaben definiert, welche eine bestmögliche (d.h. vollständige) Extraktion einzelner Zeichen sicherstellen sollten:

- Verwendung eines weißen bzw. hellen Papiers
- Horizontale Ausrichtung
- Einzeilig
- Rechtsläufige Schreibrichtung

- Klar abgegrenzte, nicht überlappende Buchstaben bzw. Ziffern
- (Möglichst) starke Linienführung mit dunkler Farbe

Ein so (handschriftlich) erfasster Text wird per Scanner digitalisiert.

5.2.2 Ablauf

Die Bearbeitung des digital vorliegenden Bilds zum Zwecke der Extraktion einzelner, durch ein KNN zu klassifizierender Zeichen erfolgt nach folgenden Schritten:

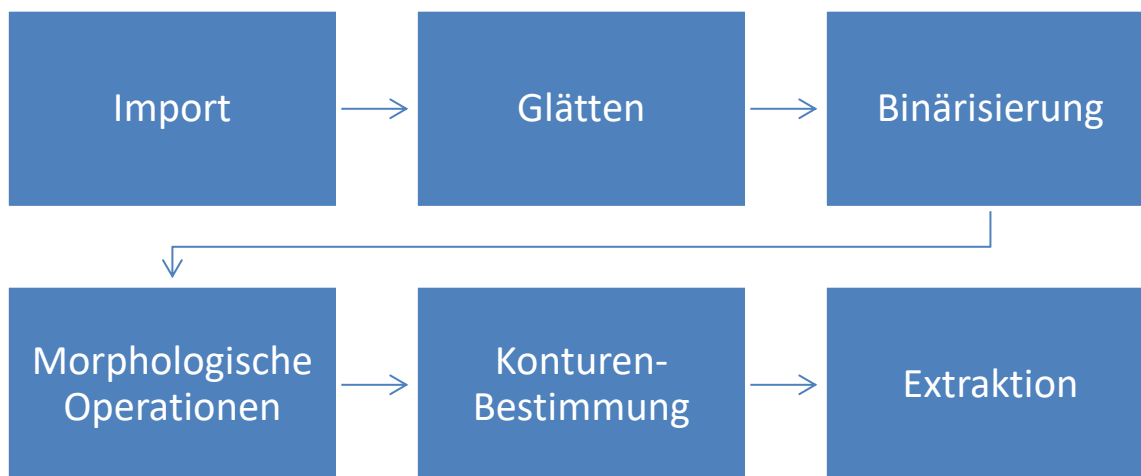


Abbildung 26 Übersicht Bildvorverarbeitung

5.2.3 Umsetzung

Die folgende Umsetzung folgt der (theoretischen) Beschreibung aus Kapitel 2 und stellt im groben die verwendeten Funktionen aus der freien Programmierbibliothek OpenCV (Open Computer Vision) (s. [OCV]) vor.

Dabei handelt es sich um eine, ursprünglich von Intel entwickelte, Algorithmen-Sammlung für Bildverarbeitung und maschinelles Sehen, welche unter der BSD-Lizenz für verschiedene Plattformen (u.a. Windows, Linux, MacOS) vertrieben wird.

Eine Anleitung zur Einrichtung von OpenCV in Eclipse findet sich im Anhang.

Der vollständige Code findet sich in der Klasse GlyphExtraction (Projekt opencv). Diese ist eigenständig lauffähig und wird für die Extraktion einzelner Zeichen aus einem Bild eingesetzt.

5.2.3.1 Einlesen des Bildes

Die Repräsentation von u.a. Bildern folgt in OpenCV durch die Klasse Mat. Das Laden eines Bildes erfolgt mit dem Aufruf

```
Mat src = Imgcodecs.imread("C:/test/iban_handwritten.png", 0);
```

Listing 1 OpenCV: Import eines Bildes

Der erste Parameter definiert den Pfad zur Bild-Datei, über das Flag 0 als zweiten Parameter wird das (gewünschte) Auslesen der Bild-Datei als Graustufenbild erreicht.

OpenCV unterstützt (unter Windows) die gängigsten Bildformate (BMP, JPG, PNG, TIFF).

5.2.3.2 Glätten

Wie in Kapitel 2.2 erwähnt wird Glätten zur Reduktion von Bilderrauschen eingesetzt. In OpenCV wird hierfür einfach folgende Methode aufgerufen:

```
Imgproc.blur(src, dest, new Size(6, 6));
```

Listing 2 OpenCV: Glätten eines Bildes

Der Aufruf von `blur` liefert (gleich dem Aufruf der meisten Methoden aus der Klasse `Imgproc`) keinen Rückgabewert, entsprechend muss das Ziel (vom Typ `Mat`) der Bildmanipulation als 2. Parameter übergeben werden. Mit `Size(6, 6)` wird die Größe des eingesetzten Filters (in diesem Fall 6x6 Pixel) festgelegt.

5.2.3.3 Binärisierung

Die Binärisierung des geglätteten Bildes erfolgt (bei Angabe eines globalen Schwellwertes) mit folgendem Aufruf:

```
Imgproc.threshold(dest, dest, 200, 255, Imgproc.THRESH_BINARY_INV);
```

Listing 3 OpenCV: Binärisierung eines Bildes

Erster und Zweiter Parameter definieren wieder Quelle und Ziel der Operation (in diesem Fall wird das Bild überschrieben), der dritte Parameter definiert den Schwellwert, der vierte den zu setzenden Wert. Nachdem mit dem 5. Parameter angeführt wird, dass das invertierte Schwellwert-Verfahren zum Einsatz kommen soll, werden alle Pixel mit einem Graustufen-Wert kleiner 200 auf den Wert 255 (Weiß) gesetzt, die restlichen auf 0 (Schwarz). Dies scheint im ersten Moment ungewöhnlich, dient jedoch als Vorbereitung für die Konturen-Bestimmung.

5.2.3.4 Öffnen und Schließen

Öffnen entspricht der Hintereinander-Ausführung von Erosion und Dilatation:

```
Imgproc.erode(dest, dest, Imgproc.getStructuringElement(Imgproc.MORPH_RECT, new Size(2, 2)));  
Imgproc.dilate(dest, dest, Imgproc.getStructuringElement(Imgproc.MORPH_RECT, new Size(2, 2)));
```

Listing 4 OpenCV: Öffnen-Operation

Wiederum erfolgt ein Überschreiben des bestehenden Bilds, als 3. Parameter wird ein Rechteck-Filter mit der Größe 2x2 definiert.

Die Schließen-Operation lässt sich durch die vertauschte Ausführung der oberen beiden Funktionen (d.h. zuerst dilate, anschließend erode) erreichen. Die Parameter bleiben dabei die gleichen.

5.2.3.5 Konturen-Bestimmung

Der Aufruf zur Berechnung der Konturen im Binärbild wirkt auf den ersten Blick unspektakulärer als er tatsächlich ist:

```
Imgproc.findContours(tmp, contours, hierachy, Imgproc.RETR_EXTERNAL, Img-
proc.CHAIN_APPROX_SIMPLE);
```

Listing 5 OpenCV: Konturen-Bestimmung eines Bildes

Entgegen der bisherigen Gewohnheit wird nur eine Kopie tmp (erstellt über die copyTo-Methode von Mat) übergeben, da das Bild durch den Funktionsaufruf verändert wird. Die OpenCV-Implementierung des zugrundeliegenden Algorithmus zur Konturen-Bestimmung betrachtet nun allein jene Pixel, deren Wert ungleich 0 ist – was das zuvor verwendete, invertierte Schwellwertverfahren erklärt. Die ermittelten Konturen werden als Liste von MatOfPoint-Objekten (eine Unterklasse von Mat) in der Variable contours abgelegt.

Parameter 3 und 4 betreffen die Hierarchie (s. [OCV-c]) erkannter Konturen. Mit der speziellen Wahl des vierten Parameters wird festgelegt, dass allein äußere Konturen gesucht sind. Der dritte Parameter muss zwar übergeben werden, die entsprechende Variable wird aber nicht weiter verwendet.

Der fünfte Parameter bestimmt, dass Konturen möglichst platzsparend abgelegt werden sollen: grundsätzlich umfassen diese nämlich die Menge aller auf der Konturform liegenden Punkte, in vielen Fällen ist jedoch eine Vereinfachung möglich (wenn die Kontur bspw. einer Geraden ähnelt, müssen nur mehr die beiden Eckpunkte gespeichert werden).

Zur weiteren Vereinfachung wird die Konturenhülle mit Hilfe des Douglas-Peucker-Algorithmus (s. Kapitel 2.5.1) näherungsweise beschrieben. Hierfür wird über alle erkannten Konturen iteriert und folgender Code aufgerufen:

```
contours.get(i).convertTo(mMOP2f, CvType.CV_32FC2);
Imgproc.approxPolyDP(mMOP2f, mMOP2f, 2, true);
mMOP2f.convertTo(contours.get(i), CvType.CV_32S);
```

Listing 6 OpenCV: Approximation der Konturenhülle

Die eigentliche Reduktion wird in Zeile 2 angestoßen, wobei der 3. Parameter die gewünschte Näherung ε angibt und der vierte bestimmt, ob Anfangs- und Endpunkt der Kontur verbunden sein müssen.

Die Konvertierungen der ersten und dritten Zeile sind notwendig, da der Aufruf von `approxPolyDP` die Klasse `MatOfPoint2f` voraussetzt, welche Elemente vom Typ `float`, anstelle von Elementen des Typs `int` in `MatOfPoint` führt.

5.2.3.6 Zeichen-Extraktion und -Speicherung

In einem weiteren Schritt wird zuerst eine Untergrenze der Fläche erkannter Konturen bestimmt: für die weitere Verarbeitung müssen Konturen mindestens 10% der größten Kontur umfassen. Die Fläche lässt sich mit Hilfe der Methode `Imgproc.contourArea` bestimmen.

Zusätzlich wird mit `boundingRect` jenes minimal die entsprechende Kontur umschließende Rechteck bestimmt und dieses in eine Liste abgelegt. Die Liste wird danach anhand der horizontalen Ausrichtung der Rechtecke sortiert, womit sich eine Orientierung gleich der handschriftlichen Vorlage ergibt.

Das bis zur Bestimmung der Konturen bearbeitete Bild wird anschließend invertiert, um wieder schwarze Schrift auf weißem Hintergrund zu haben und damit deckungsgleich mit den KNN-Trainingsdaten zu sein.

Nun kann über die Liste der zuvor berechneten und sortierten Rechtecke iteriert werden um mit den Koordinaten derselben die Zeichen-Extraktion zu ermöglichen. Hierzu wird einfach die Methode `submat` auf dem zuvor invertierten Bild, mit dem Rechteck-Objekt als Parameter, aufgerufen.

Die ersten Versuche haben dabei zwei Stolpersteine offenbart: Zum einen werden bei der Bestimmung der die Konturen umschließenden Rechtecke teilweise Ränder ungünstig über die Zeichen gelegt, wodurch einzelne Aspekte der Zeichen gar nicht extrahiert werden können:



Abbildung 27 OpenCV: Berechnetes Rechteck liegt über Zeichen

Dieses Problem kann durch eine nachträgliche Vergrößerung der Rechtecksfläche sowie Verschiebung der Koordination der linken oberen Ecke behoben werden:

```
rect.width += 3;  
rect.height += 3;  
rect.x -= 2;  
rect.y -= 2;
```

Listing 7 OpenCV: Nachbearbeitung der die Konturen umfassenden Rechtecks-Maße

Des Weiteren müssen die extrahierten Zeichen auf eine den Trainingsdaten bzw. den Eingangsneuronen des KNN kompatible Auflösung gebracht werden. Wird hier nicht den Seitenverhältnissen der ursprünglich handerfassten Zeichen Rechnung getragen, kann es zu unerwünschten und störenden Verzerrungen kommen, was die Erkennungsrate deutlich verringert.

Zum Schluss müssen die Bilder zu den einzelnen, erkannten Zeichen nur mehr gespeichert werden:

```
Imgcodecs.imwrite(filename.substring(0, filename.lastIndexOf(".")) + "_" + i +  
".png", t);
```

Listing 8 OpenCV: Export eines Bildes

Zur eindeutigen Unterscheidung wird einfach ein fortlaufender Index im Dateiname gesetzt, womit über den Index auch die Links-Rechts Ausrichtung der einzelnen Zeichen zwecks späterer Auswertung nachvollziehbar wird.

5.2.4 Parameter-Wahl

Ein Großteil der obigen Parameter-Werte wurde durch Versuch und Irrtum bestimmt. Außerdem wurde durch die eingangs gestellten Anforderungen an die handschriftliche Erfassung einer IBAN schon etliches an Komplexität genommen, was etliche, ansonsten zu beachtenden, Eventualitäten bereits im Vorhinein ausgeschlossen haben.

5.3 KNN-Eigenentwicklung

5.3.1 Hintergrund

Inspiziert von [Nie] wo u.a. die kompakte Entwicklung eines KNN auf Basis der Programmiersprache Python unter Einbindung von NumPy, einer Python-Programmbibliothek für die einfache und effiziente Handhabung von Matrizen und Vektoren, präsentiert wurde, war eine äquivalente, ebenso grundsätzliche Implementierung in Java angestrebt.

Auch wenn insbesondere in der Einarbeitungsphase zum Themenkomplex neuronaler Netze die (Aus-)Programmierung simpler, jedoch fundamentaler Eigenschaften eben jener Netze Erkenntnisgewinne und damit eine Hilfestellung fürs Verständnis derselben liefern konnten, haben sich dennoch schnell etliche Unzulänglichkeiten gezeigt, welche die Grenzen der Eigenentwicklung offenbart und damit den Einsatz eines bereits verfügbaren Frameworks nahegelegt haben.

Die folgende Beschreibung sollte deshalb auf Basis des entsprechenden Quellcodes (auszugsweise) darstellen, wie die Umsetzung erfolgte, wo die Stolpersteine lagen sowie die Entscheidung hin zur Verwendung eines externen Frameworks nachvollziehbar machen.

5.3.2 Ablauf

Umgesetzt wurde ein voll-verknüpftes FFN mit freier Wahl der Anzahl der Schichten, der Anzahl Neuronen pro Schicht, Epochen-Anzahl, Batch-Größe und Lernrate, welches sich auf das Erlernen, auf Basis der logistischen Funktion als Aktivierungs- und der quadratischen Kostenfunktion als Fehlerfunktion, des MNIST-Datensatzes beschränkt.

Hierzu durchläuft die Anwendung mehrere Schritte:

- Optional werden Trainings- und Testdaten aus dem MNIST-Datensatz gelesen und im PNG-Format zur Verfügung gestellt. Dieser Schritt muss grundsätzlich nur einmal durchlaufen werden, da die konvertierten Bilder dauerhaft abgelegt werden können.
- Anschließend werden die Anzahl der Schichten sowie die Anzahl der Neuronen pro Schicht definiert.
- Das Netz-Training, auf Basis der zuvor definierten Trainingsdaten, erfolgt mit Hilfe des Stochastischen Gradientenverfahrens, optional kann eine Validierung anhand der Testdaten erfolgen.

Die Ablaufsteuerung erfolgt direkt im Quellcode durch Auskommentierung entsprechender Code-Zeilen.

5.3.3 Umsetzung

5.3.3.1 Basis

Die Implementierung erfolgt im Projekt nnetwork und allein mit den Bordmitteln von Java 8, d.h. ohne Einbeziehung von 3rd-party-Libs.

Dem Klassenkonzept von Java folgend werden sowohl die Info zum gesamten KNN als auch zu den einzelnen Netz-Schichten in einzelne Klassen (NNParam bzw. NNLayer) abgelegt, gleiches gilt auch für die Trainings- bzw. Testdaten (NNTraining), welche eine Sammlung von Eingabedaten (vom Typ NNInput) darstellen. Letztere halten zu jedem Bild Dateipfad als auch erwartetes Klassifizierungs-Ergebnis. Das Ergebnis wird als Liste von 10 Double-Werten abgebildet: hierbei definiert ein Wert 1 zum Listen-Index n , dass das Bild die Ziffer n darstellt, ansonsten wird 0 gesetzt.

Die notwendigen Methoden zur Berechnung der Netz-Gewichte werden in einer eigenen Klasse NNFunctions zur Verfügung gestellt, zusätzlich stehen in NNHelper u.a. funda-

mentale Operationen zum Lesen der (Eingabe-)Bilder zur Verfügung sowie in NNInit Methoden zum (initialen) Befüllen der Netz-Gewichte.

5.3.3.2 Initialisierung

NNMain definiert den Rahmen der Applikation und enthält auch die entsprechende main-Methode.

Die Anzahl der zu verwendeten Trainings- (1. Parameter) bzw. Test-Daten (2. Parameter), welche in Kapitel 5.1.3.1 bereitgestellt wurden, werden wie folgt definiert:

```
NNTraining nnTraining = NNHelper.getInputs(6000, 10000, false);
```

Listing 9 NNetwork: Definition der Trainings- und Test-Daten

Der dritte Parameter, sofern gesetzt, ermöglicht die zufällige Wahl eben dieser Bilder für den Fall dass die Anzahl der gewünschten Bilder kleiner als die Anzahl der verfügbaren wäre.

Die Klasse NNInit definiert die verschiedenen Netz-Schichten:

```
nnInit.init(sizes);
```

Listing 10 NNetwork: Initialisierung der Schichten

Hierbei ist sizes eine Liste von Integer-Werten, wobei die Listen-Größe die Anzahl der Schichten (inklusive Ein- und Ausgabe-Schicht) bzw. der jeweilige Wert die Anzahl der Neuronen pro Schicht festlegt.

Der Aufruf initialisiert nicht nur die Objekte vom Typ NNLayer sondern legt auch die Verknüpfungs-Gewichte zwischen den Neuronen (abgebildet als Liste einer Liste von Double-Werten) zweier Schichten zufällig fest.

Als Besonderheit werden die Bias-Werte zu einer Schicht gesondert (als Liste von Double-Werte) geführt: dies deshalb, da die Berechnung der Bias-Änderung während des Trainings vereinfacht und gesondert erfolgen kann (vgl. [Nie], Chapter 2). Im Zuge der Initialisierung werden auch die Bias-Werte zufällig gesetzt.

5.3.3.3 Training

Das Training des FFN wird mit folgendem Aufruf gestartet:

```
NNFunctions.SGD(nnInit.getNnParams(), nnTraining.getTrainingData(), 200, 10, 0.1, nnTraining.getTestData());
```

Listing 11 NNetwork: Aufruf des Stochastischen Gradientenverfahrens

Der erste Parameter übergibt das zuvor definierte Netzwerk, der zweite bzw. (optionale) sechste Parameter die Trainings- bzw. Test-Daten. Der dritte Parameter definiert die Anzahl der Trainings-Epoche, der vierte die Batch-Größe und der fünfte die (fixe) Lernrate.

In der Methode `NNFunctions.SGD` werden zuerst die Trainingsdaten zufällig gemischt, diese anschließend in Stapel der definierten Größe unterteilt und auf diese einzeln (mit Lernrate `eta`) die Gewichts- bzw. Bias-Änderungen angewendet:

```
updateMiniBatch(nnParams, miniBatch, eta);
```

Listing 12 NNetwork: Gewichts-/Bias-Änderungen auf einzelne Batches

Dabei wird für jedes Trainings-Beispiel zuerst der Eingabewert bestimmt:

```
List<Double> activation = NNHelper.getInputFromPath(nnInput.getInput());
```

Listing 13 NNetwork: Bestimmung der Bildpixel-Grauwerte

Die Liste der Double-Werte enthält für jeden Bild-Pixel, von links oben nach rechts unten, den entsprechenden Grauwert (mit den Grenzen 0 = Weiß bzw. 1 = Schwarz). Die Berechnung des Grauwerts basiert dabei auf der Formel aus Kapitel 2.3.

Anschließend werden, Kapitel 3.4.2 folgend, die Werte der Ausgabeschicht berechnet, indem die Werte der Eingabeschicht iterativ über alle verdeckten Schichten propagiert werden:

```
for (int i = 1; i < nnParams.getNumLayers(); i++) {
    NNLayer nnLayer = nnParams.getNnLayers().get(i);

    List<Double> z = new ArrayList<Double>();
    for (int j = 0; j < nnParams.getSizes().get(i); j++) {
        z.add(calcInput(activation, nnLayer.getWeights().get(j), nnLayer.getBiases().get(j)));
    }
    zs.add(z);

    activation = new ArrayList<Double>();
    for (Double a : z) {
        activation.add(sigmoid(a));
    }
    activations.add(activation);
}
```

Listing 14 NNetwork: Berechnung der Netz-Schicht-Ausgaben

Hierbei berechnet `calcInput` den Netto-Input der Aktivierungsfunktion und `sigmoid` die korrespondierende Ausgabe nach Anwendung derselbigen.

Nach Durchlaufen der äußeren Schleife ist die Ausgabe des FFN verfügbar, womit die Abweichung zur gewünschten Ausgabe des Trainings-Bildes in der Methode `NNFunctions.backprop` berechnet und damit auch der entsprechende Fehler δ_k der Ausgabeschicht bzw. δ_j der verdeckten Schicht(en) bestimmt werden kann. Damit lassen sich nun die notwendigen Gewichts- und Bias-Änderungen ableiten (letztere sind einfach dem entsprechenden Fehler δ eines Neurons äquivalent).

Nach vollständigem Durchlaufen eines Batches werden die so bestimmten Änderungen in die Gewichts- bzw. Bias-Werte des FFN übernommen:

```

for (int i = 0; i < nnParams.getNumLayers(); i++) {
    NNLayer layer = nnParams.getNnLayers().get(i);
    NNLayer backpropLayer = backpropLayers.get(i);

    // biases
    for (int j = 0; j < layer.getBiases().size(); j++) {
        layer.getBiases().set(j, layer.getBiases().get(j) - (eta / mini-
Batch.size()) * backpropLayer.getBiases().get(j));
    }

    // weights
    for (int j = 0; j < layer.getWeights().size(); j++) {
        List<Double> layerWeight = layer.getWeights().get(j);
        List<Double> backpropLayerWeight = backpropLayer-
er.getWeights().get(j);

        for (int k = 0; k < layerWeight.size(); k++) {
            layerWeight.set(k, layerWeight.get(k) - (eta / mini-
Batch.size()) * backpropLayerWeight.get(k));
        }

        layer.getWeights().set(j, layerWeight);
    }

    nnParams.getNnLayers().set(i, layer);
}

```

Listing 15 NNetwork: Schreiben der Gewichts-/Bias-Änderungen

Dies wird nun mit allen übrigen Batches wiederholt, um am Ende einer Epoche, sofern noch nicht die Anzahl der gewünschten Epochen erreicht wurde, eine neue, zufällige Zusammenstellung von gestapelten Trainings-Daten zu bestimmen und einen neuen Durchlauf zu starten.

5.3.3.4 Validierung

Sofern Testdaten initial übergeben wurden wird anhand dieser am Ende jeder Trainings-Epoche die Erkennungsrate des neu justierten Netzes geprüft:

```

System.out.println("Epoch " + e + ": " + evaluate(nnParams, testData) + " / " +
testData.size());

```

Listing 16 NNetwork: Netz-Erkennungsrate auf Basis der Testdaten

Hierfür werden die Testdaten, welche nicht Basis des Trainings waren, in das FFN eingespeist, der Index mit dem höchsten Double-Wert in der Ausgabeschicht bestimmt und dieser mit dem erwarteten Klassifizierungsergebnis verglichen. Stimmen die entsprechenden Indizes überein, konnte das Netz die am Bild dargestellte Ziffer richtig erkennen.

5.3.4 Probleme

Bereits während der initialen Implementierungsphase und nach den ersten Tests haben sich etliche Schwierigkeiten gezeigt, welche die weitere Umsetzung einer Eigenentwicklung in Frage stellten.

5.3.4.1 Performance

Aus praktischen Gründen (einfacheres Handling und bessere Übersichtlichkeit) wurden nicht Arrays mit `double`-Werten, sondern (tlw. mehrfach verschachtelte) `ArrayList`-Objekte zur Verwaltung der Gewichts- und Bias-Werte verwendet. Dies schlägt sich in (leicht) erhöhten Zugriffszeiten bzw. einem höheren Speicherverbrauch nieder (vgl. [StO-c]), was in Summe eine längere Laufzeit bei Training und Anwendung des Neuronalen Netzes bedingt.

Gleichfalls werden die Trainings- und Test-Bilder als Listen von Pixel-Grauwerten (vom Typ `Double`) und damit (permanent) im Speicher geführt. Dies kann sich als Flaschenhals (bis hin zu einem Out-of-Memory) bei Verwendung einer höheren Anzahl von Input-Daten erweisen, womit auch eine (vernünftige) Skalierung der Implementierung in Zweifel gezogen werden kann.

Auch die Unterstützung mehrerer CPUs (Multi-Threading) oder gar die Verwendung einer GPU zum Zwecke der Parallelisierung der notwendigen Rechenschritte (und einer damit einhergehenden Reduzierung der Laufzeit) ist alles andere als banal, bedarf entsprechenden (Vor-)Wissens und wäre nur durch einen relativ großen Ressourceneinsatz zu erreichen – selbst das nicht garantiert.

5.3.4.2 Neue Features

Der (evolutionäre) Ansatz, das KNN schrittweise und parallel zur Einarbeitung in die Thematik zu entwickeln, war zwar, wie erwähnt, hilfreich für ein fortschreitendes Verständnis der Arbeitsweise von KNNs, gleichzeitig jedoch auch ein überraschend langwieriges (Trial-and-Error-)Verfahren bis zum Erhalt (halbwegs) brauchbarer Ergebnisse, denen zudem auch noch ein enger Rahmen gesetzt war (z.B. fix vorgegebene KNN-Ausprägung oder bestimmte Aktivierungsfunktion).

Der Versuch einer entsprechenden Modularisierung bzw. Erweiterung oder gar Optimierung der Implementierung bedingt jedoch rasch ein (schlimmstenfalls wiederholtes) Refactoring am bestehenden Code, womit wiederum der (zusätzliche) zeitliche Aufwand unverhältnismäßig stark ansteigt.

5.3.4.3 Komplexität

Der den Neuronalen Netzen innewohnende Ansatz, durch Training zu Lernen, führt unweigerlich auch zur Frage, wann von einem „guten“ Training die Sprache sein kann. Ein

KNN verhält sich deshalb insofern als „Black Box“, da unmittelbar (d.h. ohne Vorwissen oder entsprechende Vergleichsmöglichkeit - und damit rein intuitiv) nur ein vager Richtwert für bspw. die gewünschte Erkennungsrate angeführt werden kann. Weicht die gemessene Performance des KNN von diesem Richtwert ab, ist auch nicht unmittelbar klar, wo die Ursache liegt: diese könnte sowohl durch die verwendeten Trainings- oder Test-Daten begründet sein, aber auch die Folge eines Code-Fehlers sein.

Im Falle letzteres ist ein Debugging aber alles andere als trivial, nicht zuletzt aufgrund der Komplexität im Ablauf (Handling vieler Einzel-Werte, verteilt über mehrere Klassen) sowie der fehlenden bzw. stiefmütterlich behandelten Optimierung im Quellcode

Auch ist nicht auszuschließen, dass aufgrund der i.d.R. kleineren Nutzerbasis einer Eigenentwicklung manche Fehler gar nie auffallen.

5.4 Deeplearning4j

Wie im vorherigen Kapitel erwähnt ist die Verwendung eines speziellen Frameworks bei Implementierung eines KNN mehr als ratsam: diese bieten nicht nur einen großen Funktionsumfang sondern sind zudem auch (deutlich) performanter und weniger fehleranfällig (da von deutlich mehr Benutzern verwendet) als eigene Umsetzungen (insbesondere mit dem doch beschränkten Wissen eines Anfängers).

Zwar findet sich eine ganze Reihe, größtenteils sogar frei-verfügbarer Bibliotheken (vgl. [Wik-d]), omnipräsent scheinen Googles TensorFlow (s. [Std15]) oder Facebooks Caffe (s. [Hei17]) zu sein, die ursprüngliche Recherche nach einem NumPy-Ersatz für Java hat jedoch auf Deeplearning4j (Deep Learning for Java, auch DL4J, s. [DL4]) geführt.

Dabei handelt es sich um ein Open-Source Framework für künstliche Intelligenz bzw. maschinelles Lernen, programmiert für die JVM (nahezu ein Alleinstellungsmerkmal) und u.a. mit Unterstützung verschiedenster, teilweise bereits in Kapitel 3 beschriebener Techniken zur Optimierung von Neuronalen Netzen. Dies macht Deeplearning4j zu einem mehr als brauchbaren Gegenstand einiger Untersuchungen zur Performance von Neuronalen Netzen bzw. den Einfluss diverser Adaptierungen der Hyperparameter.

Deeplearning4j, unter den Top 10 der Deep-Learning-Projekte auf Github (vgl. [May]), wird maßgeblich von der in San Francisco beheimateten Firma SkyMind (die auch entsprechenden kommerziellen Support liefert, s. [Sky]) entwickelt.

Bei Erstellung der Diplomarbeit wurde die Version 0.8.0 von Deeplearning4j herangezogen.

5.4.1 Basis

Deeplearning4j basiert auf ND4J (N-Dimensional Arrays for Java, s. [ND4]), einer in C++ geschriebenen und auf der JVM (Java Virtual Machine) laufenden Bibliothek, spezialisiert auf Lineare-Algebra- resp. Matrix-Operationen - damit ideal für die notwendigen Rechenschritte (und darauf bauenden Konzepten) bei Neuronalen Netzen.

Dabei liefert die ND4J-API Wrapper für verschiedene BLAS-Versionen (Basic Linear Algebra Subprograms, s. [BLA]), welche elementare Routinen zu eben jenen Lineare-Algebra-Operationen spezifizieren.

ND4J abstrahiert zudem das zugrundeliegende Backend: Berechnungen können auf einer (bzw. mehreren) CPU(s) oder auf einer GPU (Graphics Processing Unit) laufen. Gerade letztere, oft Bestandteil von Grafikkarten, eignen sich ob ihrer massiv-parallelen Struktur (vgl. [NVI]) hervorragend für die Anwendung bei Neuronalen Netzen, da die dahinterliegenden Algorithmen (relativ) einfache Berechnungen auf vielen verschiedenen Netzknoten vorsehen. Die GPU-Unterstützung von ND4J ist aktuell auf CUDA (Compute Unified Device Architecture), eine von Nvidia entwickelte Technik für die Abarbeitung von Programmteilen durch die GPU und damit auf Hardware dieses Herstellers beschränkt.

5.4.2 Umsetzung

Die folgende Beschreibung sollte als Ergänzung zum Quellcode im Projekt dl4j betrachtet werden und stellt entsprechend keine vollständige Aufzählung aller (notwendigen) Schritte dar.

Zur Einrichtung von DL4J sei auf die entsprechende Setup-Anleitung im Anhang verwiesen.

5.4.2.1 Initialisierung

In der Klasse MutliLayerNetwork führt DL4J ein entsprechendes KNN, dessen allgemeine Eigenschaften sich über `NeuralNetConfiguration.Builder` konfigurieren lassen:

```
NeuralNetConfiguration.Builder nccBuilder = new NeuralNetConfigurati-
on.Builder();
// (reproduzierbare) Initialisierung der Gewichts-Werte
nccBuilder.seed(seed);
// Aktivierung und Verwendung der L2-Regularisierung
nccBuilder.regularization(true);
nccBuilder.l2(0.0005);
// variable Lernrate
nccBuilder.learningRateDecayPolicy(LearningRatePolicy.Schedule);
nccBuilder.learningRateSchedule(lrSchedule);
// Gewichts-Initialisierung mit Xavier
nccBuilder.weightInit(WeightInit.XAVIER);
// Verwendung des Stochastischen Gradientenverfahrens
```

```
nccBuilder.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
// Verwendung eines Momentums
nccBuilder.updater(Updater.NESTEROVS);
nccBuilder.momentum(0.9);
```

Listing 17 DL4J: KNN-Konfiguration (Auszug)

Die variable Lernrate wird als Map definiert:

```
Map<Integer, Double> lrSchedule = new HashMap<>();
lrSchedule.put(0, 0.01);
lrSchedule.put(1000, 0.005);
lrSchedule.put(3000, 0.001);
```

Listing 18 DL4J: Variable Lernrate

Dabei definiert pro Eintrag der erste Parameter ab welcher Batch-Einheit die jeweilige Lernrate (zweiter Parameter) gilt.

5.4.2.2 Layers

DL4J unterstützt u.a. mit ConvolutionalLayer, SubsamplingLayer, DenseLayer und OutputLayer verschiedene Schicht-Typen, welche bspw. in einem CNN zur Anwendung kommen. Die jeweilige Instanzierung erfolgt ähnlich, je Typ jedoch mit unterschiedlichen Parametern:

```
ConvolutionLayer c11 = new ConvolutionLayer.Builder().kernelSize(5,
5).stride(1, 1).nIn(1).nOut(20)
    .activation(Activation.IDENTITY).build();
SubsamplingLayer s11 = new SubsamplingLayer.Builder().kernelSize(2,
2).stride(2, 2)
    .poolingType(SubsamplingLayer.PoolingType.MAX).build();
```

Listing 19 DL4J: ConvolutionalLayer und SubsamplingLayer

kernelSize bzw. stride definieren die Filtergröße bzw. Schrittlänge, nOut die Anzahl der Feature-Maps sowie activation bzw. poolingType die gewünschte Aktivierungs- bzw. Pooling-Funktion.

Nach wiederholtem Setzen von ConvolutionalLayer und SubsamplingLayer folgen abschließend eine vollverknüpfte Standard- (mit der Aktivierungsfunktion Rectified Linear Unit und einem Dropout von 50%) sowie die Ausgabeschicht:

```
DenseLayer dl = new DenseLayer-
er.Builder().activation(Activation.RELU).nOut(500).dropout(0.5).build();
OutputLayer ol = new OutputLay-
er.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD).nOut(outputNum)
    .activation(Activation.SOFTMAX).build();
```

Listing 20 DL4J: DenseLayer und OutputLayer

In der Ausgabeschicht wird die (negative) Log-Likelihood-Funktion als Kostenfunktion (vgl. [StE-d]) und für die Aktivierung die Softmax-Funktion gesetzt (die Verwendung letzterer führt u.a. dazu, dass die Summe über alle Ausgabe-Werte exakt 1 entspricht) – dies kann

als Äquivalent zur Kombination aus quadratischer Kostenfunktion und logistischer Funktion gesehen werden, liefert jedoch eine verbesserte Lernleistung des KNN (vgl. [Nie], Chapter 3).

Die so definierten Schichten werden, unter Angabe des gewünschten Schicht-Index und ergänzt um die Definition der Eingabe-Dimension sowie der Aktivierung der Backpropagation, kombiniert:

```
ListBuilder listBuilder = nccBuilder.list();
listBuilder.layer(0, cl1);
listBuilder.layer(1, sl1);
listBuilder.layer(2, cl2);
listBuilder.layer(3, sl2);
listBuilder.layer(4, dl);
listBuilder.layer(5, ol);
listBuilder.setInputType(InputType.convolutionalFlat(28, 28, 1));
listBuilder.backprop(true);
```

Listing 21 DL4J: Schichten-Kombinierung

Damit kann nun auch ein `MultiLayerNetwork` initialisiert werden:

```
MultiLayerNetwork model = new MultiLayerNetwork(listBuilder.build());
model.init();
```

Listing 22 DL4J: Initialisierung `MultiLayerNetwork`

5.4.2.3 Training

Wie in Kapitel 5.1.3.2 beschrieben liegen die Trainings-Daten (gleich den Test-Daten) gruppiert nach Zeichen in Ordnern mit der Bezeichnung des jeweiligen Zeichen-ASCII-Codes vor. DeepLearning4j nutzt diese Struktur und ermöglicht eine Klassifizierung anhand der Ordner-Bezeichnung, womit auch bei der späteren Anwendung des KNN eine einfache Zeichen-Zuordnung möglich wird.

Die Trainingsbilder werden mit folgendem Code geladen:

```
FileSplit train = new FileSplit(trainData, NativeImageLoader.ALLOWED_FORMATS,
    randNumGen);
ImageRecordReader recordReader = new ImageRecordReader(height, width, 1, new
    ParentPathLabelGenerator());
recordReader.initialize(train);
```

Listing 23 DL4J: Laden der Trainingsbilder

`FileSplit` listet alle Dateien aus dem über `trainData` referenzierten Ordner, mit `ImageRecordReader` werden die so ermittelten Dateien als Bilder (der Größe `height` x `width` und mit einem Grauwert-Farbkanal) geladen, wobei `ParentPathLabelGenerator` definiert, dass die entsprechenden Ordner-Bezeichnungen als gewünschte Ausgabe herangezogen werden.

Die Bild-Grauwerte werden anschließend skaliert (womit jeder Pixel einen Wert zwischen 0 und 1, d.h. Schwarz und Weiß erhält) und in Training-Batches der Größe `batchSize` unterteilt mit 62 möglichen Klassifizierungs-Ergebnissen:

```
DataSetIterator dataIter = new RecordReaderDataSetIterator(recordReader,
batchSize, 1, 62);
DataNormalization scaler = new ImagePreProcessingScaler(0, 1);
scaler.fit(dataIter);
dataIter.setPreProcessor(scaler);
```

Listing 24 DL4J: Definition der Training-Batches

Das Training wird anschließend für die gewünschte Anzahl an Epochen angestoßen:

```
for (int i = 0; i < nEpochs; i++) {
    model.fit(dataIter);
}
```

Listing 25 DL4J: Training des KNN

5.4.2.4 Validierung

Die Validierung gestaltet sich ähnlich dem Training: Wiederum wird (auf Basis der Testbilder) ein `ImageRecordReader`-Objekt generiert, welches in einem `DataSetIterator` Verwendung findet. Über letzteres kann nun iteriert und je Testbild ein Vergleich der aus dem KNN bestimmten Klassifizierung (ausgeführt über die Methode `output`) mit dem erwarteten Wert angestellt werden:

```
Evaluation eval = new Evaluation(62);
while (testIter.hasNext()) {
    DataSet next = testIter.next();
    INDArray output = model.output(next.getFeatureMatrix());
    eval.eval(next.getLabels(), output);
}
```

Listing 26 DL4J: Validierung des KNN

Über den Aufruf `eval.accuracy` resp. `eval.stats` lässt sich anschließend eine kurze bzw. detaillierte Validierungsauswertung (bspw. ins Log) ausgeben.

5.4.2.5 Export und Import

Die Konfiguration sowie die (trainierten) Gewichte eines KNN lassen sich einfach mit folgendem Befehl in eine ZIP-Datei, nach erfolgreichem Abschluss der Trainingsphase, exportieren:

```
ModelSerializer.writeModel(model, locationToSave, saveUpdater);
```

Listing 27 DL4J: Export

Der dritte Parameter definiert hierbei, ob der Status des KNN-Updaters (z.B. Momentum) zwecks späterer Fortführung des Trainings mit abgelegt werden sollte.

Der Import des KNN gestaltet sich ähnlich simpel:


```
model = ModelSerializer.restoreMultiLayerNetwork(locationToSave);
```

Listing 28 DL4J: Import

5.4.2.6 GPU-Unterstützung

Aufgrund der verwendeten Hardware (integrierte Intel-CPU) steht für die Bearbeitung der Diplomarbeit keine CUDA-Schnittstelle und damit auch keine GPU-Unterstützung zur Verfügung (eine Unterstützung von Nicht-Nvidia-Hardware auf Basis von OpenCL, Open Computing Language, ist lt. Roadmap erst zu einem späteren Zeitpunkt angedacht).

Nichtsdestotrotz findet sich eine Anleitung zur entsprechenden Einrichtung im Anhang.

5.4.3 Besonderheiten

Bei Anwendung liefert das KNN ein Array mit 62 (Double-)Einträgen, wobei der jeweilige Wert die Wahrscheinlichkeit der entsprechenden Zugehörigkeit an eine bestimmte Klassifizierungs-Klasse angibt. Die Reihenfolge der verschiedenen Klassifizierungen kann jedoch nicht einfach als aufsteigender ASCII-Wert der bestimmbaren Zeichen angenommen werden, sondern wird in der DL4J-Klasse `FileSplit` über `FileUtils.listFiles` während der Trainingsphase bestimmt (und kann bspw. dazu führen, dass 3-stellige Ordnerbezeichnungen vor 2-stelligen angeführt werden). Um deshalb unabhängig von einem Vorliegen der Trainingsbilder eine korrekte Zuordnung zwischen Klassifizierungs-Ergebnis des KNN und passendem Zeichen zu erhalten (und damit konform mit dem trainierten Netz zu sein), ist es notwendig die entsprechende Reihenfolge wegzusichern.

Dies wird mit folgendem Code erreicht, welcher nach Training des KNN ausgeführt wird:

```
List<Integer> labelsTmp = new ArrayList<Integer>();
File[] directories = trainData.listFiles(File::isDirectory);
for (File d : directories) {
    labelsTmp
        .add(Integer.valueOf(d.getAbsolutePath().substring(d.getAbsolutePath().lastIndexOf("\\") + 1)));
}
```

Listing 29 DL4J: Bestimmung der Klassifizierungs-Klassen-Reihenfolge des KNN

Die so bestimmten Listen-Werte könnten bspw. in einer CSV-Datei weggesichert und bei Anwendung des KNN wieder herangezogen werden (in der erfolgten Umsetzung wird jedoch immer von einem Vorhandensein der entsprechenden Trainingsbilder-Ordner-Struktur ausgegangen).

5.5 Klassifizierung

Im Folgenden wird die Anwendung des in Kapitel 5.4 beschriebenen KNN auf die in Kapitel 5.2.3.6 ermittelten Bilder handschriftlich erfasster Zeichen besprochen.

5.5.1 Vorbereitung

In Kapitel 5.2.3.6 wurde der Dateiname der einzeln extrahierten Zeichen mit einem fortlaufenden Index versehen um die richtige Reihenfolge auch nachträglich sicherzustellen. Beim Auslesen der zu analysierenden Bilder ist deshalb darauf zu achten, dass die definierte Reihenfolge auch eingehalten wird.

5.5.2 Auswertung

Über die so erhaltene Liste von Bild-Dateien wird iteriert, wobei jedes Bild in das KNN eingespeist wird. Hierzu wird jede Bilddatei, auf Basis der im Framework zur Verfügung stehenden Methoden (welche wiederum tlw. auf OpenCV zurückgreifen), in ein zu DL4J kompatibles Format gebracht, gleichzeitig (den Trainingsdaten ähnlich) auch normalisiert:

```
NativeImageLoader loader = new NativeImageLoader(28, 28, 1);
INDArray image = loader.asMatrix(imageFile);
DataNormalization scaler = new ImagePreProcessingScaler(0,1);
scaler.transform(image);
```

Listing 30 DL4J: Einlesen eines zu analysierenden Bildes

Die entsprechende Klassifizierung erhält man mit folgendem Aufruf:

```
INDArray output = model.output(image);
```

Listing 31 DL4J: Analyse eines Bildes

Das Array output enthält nun den Grad der Zuordnung (zwischen 0 und 1 liegend) zu jeder durch das KNN bestimmbaren Klasse. Um das mit jeder Klasse korrespondierende ASCII-Zeichen zu bestimmen, wird die in Kapitel 5.4.3 ermittelte Liste herangezogen.

5.5.3 Interpretation

Um v.a. das (optionale) Logging übersichtlicher zu gestalten, werden nur Zuordnungsgrade mit einem (auf 2 Dezimalstellen gerundeten) Wert größer 0 herangezogen. Die Zuordnung zu einem der definierten 62 ASCII-Zeichen kann nun absteigend sortiert werden.

Exemplarisch sollte folgendes Bild interpretiert werden:



Abbildung 28 Beispiel für Bild-Analyse

Das KNN errechnet folgende Wahrscheinlichkeiten:

O: 0.56, Q: 0.2, 0: 0.13, o: 0.09, D: 0.01, q: 0.01

Sprich: das zu analysierende Zeichen ist zu 56% der (Groß-)Buchstabe O, zu 20% der (Groß-)Buchstabe Q, zu 13% die Ziffer 0, zu 9% der (Klein-)Buchstabe und zu jeweils 1% der (Groß-)Buchstabe D bzw. der (Klein-)Buchstabe q. In der Regel wird nun jenes Zeichen mit der höchsten Wahrscheinlichkeit als Ergebnis der Klassifikation herangezogen.

5.5.3.1 IBAN

Insbesondere im Falle einer IBAN muss jedoch nicht zwischen der Groß- und Kleinschreibung von Buchstaben unterschieden werden, weshalb die beiden korrespondierenden Werte addiert werden können. Programmatisch lässt sich dies relativ einfach bewerkstelligen, da der (dezimale) ASCII-Code eines Kleinbuchstaben um 32 höher liegt als der des entsprechenden Großbuchstaben:

```
for (int i = 65; i < 91; i++) {  
    double sum = Double.  
sum(output.getDouble(Arrays.asList(labels).indexOf(i)),  
        output.getDouble(Arrays.asList(labels).indexOf(i + 32)));  
    if (BigDecimal.valueOf(sum).setScale(2, RoundingMo-  
de.HALF_UP).doubleValue() > 0.0) {  
        // weiterer Code  
    }  
}
```

Listing 32 Kombinierung von Groß- und Kleinbuchstaben

Angewandt auf das vorher genannte Beispiel ergibt sich damit nun folgende Sortierung:

O: 0.65, Q: 0.2, 0: 0.13, D: 0.01

Wenn sich auch (vorerst) keine neue Reihenfolge ergeben hat, lässt sich zumindest (noch) deutlicher die Dominanz des Buchstaben O erkennen (was vermutlich der Interpretation der meisten menschlichen Beobachter entsprechen wird).

Zusätzlich lässt sich aber auch noch der Kontext eines zu analysierenden Zeichens heranziehen, um die Wahrscheinlichkeit einer richtigen Interpretation zu erhöhen. Angenommen es ist bekannt, dass das Bild das 17. Zeichens einer österreichischen (dies kann, o.B.d.A., nach funktionaler Anforderung der Diplomarbeit vorausgesetzt werden) IBAN darstellt. Nach 4.2.2.1 müsste es sich dabei um eine Ziffer handeln.

Dem kann nun dadurch Rechnung getragen werden, dass in der (sortierten) Liste aller möglichen Klassifizierungen Nicht-Ziffern übersprungen werden und somit die Ziffer mit der höchsten Wahrscheinlichkeit als neues Auswertungs-Ergebnis herangezogen wird. Im Beispiel wäre das die Ziffer 0 – womit sich doch eine Neu-Interpretation der Auswertung ergibt.

Im Falle einer (gewünschten) IBAN-Interpretation wird nun für jedes Zeichen ab der 3. Stelle die Interpretation als Ziffer vorgenommen.

Zur IBAN-Validierung kann nun der gesamte, durch das KNN interpretierte (und durch die vorherigen Schritte nachbearbeitete) Text an die entsprechende `IBAN.valueOf`-Methode (welche aus dem externen Projekt `java-iban` stammt) übergeben werden:

```
try {
    IBAN iban = IBAN.valueOf(outputText.toString());
    Log.info("valid: " + iban.isSEPA());
} catch (UnknownCountryCodeException | WrongChecksumException e) {
    Log.info("invalid IBAN: " + outputText.toString());
}
```

Listing 33 IBAN-Validierung

Fliegt keine Exception, handelte es sich um eine gültige IBAN.

6 Nachweis der Funktionalität

Im folgenden Kapitel werden einige Vergleiche zwischen der Eigenentwicklung sowie verschiedenen Implementierungen auf Basis von DL4J gezogen. Dabei wird in Grundzügen untersucht, welche Faktoren Einfluss auf relevante Performancegrößen (Laufzeit, Erkennungsrate, etc.) nehmen. Diese Messungen verfolgen u.a. den Zweck, ein robustes Neuronales Netz zu bestimmen, welches anschließend für die IBAN-Erkennung herangezogen werden kann.

6.1 Vergleich Eigenentwicklung mit DL4J

Neben dem (subjektiven) Entwicklungsaufwand, werden auch Trainings-Laufzeit und Erkennungsrate für eine Gegenüberstellung der Eigenentwicklung mit der DL4J-basierten Umsetzung (Klasse `MLPMnistTwoLayerExample`) herangezogen. Beide Implementierungen erfolgen als 3-schichtiges, voll-verknüpftes FFN, zudem werden (soweit möglich) die jeweiligen Hyperparameter aufeinander abgestimmt (und folgen den Vorgaben aus Kapitel 5.3). Die Basis für Training und Validierung bildet der MNIST-Datensatz (mit 60.000 Trainings- und 10.000 Test-Bildern).

6.1.1 Konfiguration von DL4J

Für die Definition eines entsprechenden FFN (Klasse `MLPMnistTwoLayerExample`) werden folgende 2 Layer eingebunden:

```
.layer(0, new DenseLayer.Builder().nIn(inputNum).nOut(intermediateNum).build())
.layer(1,
    new OutputLayer.Builder(LossFunction.SQUARED_LOSS).activation(Activation.SIGMOID).nIn(intermediateNum)
        .nOut(outputNum).build())
```

Listing 34 DL4J: Definition eines 3-schichtigen FFN

Die Verwendung der MNIST-Daten gestaltet sich relativ einfach, da DL4J hierfür eine eigene Klasse `MnistDataSetIterator` zur Verfügung stellt, welche sich um die Einbindung (und Normalisierung der einzelnen Pixel auf Werte zwischen 0 und 1) der entsprechenden Trainings- und Testdaten kümmert:

```
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true, rngSeed);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize, false, rngSeed);
```

Listing 35 DL4J: Integration der MNIST-Daten

`mnistTrain` und `mnistTest` (resp. die entsprechenden Aufruf von `getFeatureMatrix`) können anschließend als Parameter für die Methoden `fit` und `output` gesetzt werden.

6.1.2 Hyperparameter

Das FFN besteht in beiden Implementierungen aus 784 Eingangsneuronen, 30 Neuronen in der versteckten Schicht und 10 Ausgangsneuronen. Zur Fehlerbestimmung wird die quadratische Kostenfunktion, zur Aktivierung der einzelnen Neuronen die logistische Funktion verwendet. Als Lernrate wird der Wert 0,3 gewählt, als Epochenanzahl 100. Die Batchgröße beträgt 32.

6.1.3 Ergebnis

Auch wenn natürlich dem Umstand Rechnung getragen werden muss, dass die Verwendung von DL4J im Hintergrund etliche Abhängigkeiten zum entsprechenden Framework definiert, sticht doch der ins Auge, dass sich der Code der Eigenimplementierung auf 9 Klassen (mit entsprechender Hierarchie) erstreckt, hingegen die Implementierung auf Basis von DL4J übersichtlich innerhalb einer einzigen `main`-Methode erfolgen kann.

Zur Messung der (Trainings-)Laufzeit wird einfach die aktuelle Systemzeit vor und nach Durchlauf mit `System.currentTimeMillis` bestimmt und anschließend die entsprechende Differenz gebildet. Die Erkennungsleistung des trainierten Netzes kann mit Hilfe der entsprechenden Testdaten bestimmt (und zwischen den verschiedenen Implementierungen auch verglichen) werden.

Daraus ergeben sich folgende Messdaten (jeweils arithmetischer Mittelwert aus 3 kompletten Durchgängen):

	Eigenimplementierung	DL4J
Entwicklungsaufwand	Hoch	Einfach bis Mittel ⁴
Laufzeit	1: 13min, 8s 2: 12min, 58s 3: 14min, 52s Durchschnitt: 13min, 39s	1: 6min, 40s 2: 6min, 47s 3: 6min, 52s Durchschnitt: 6min, 43s

⁴ Abhängig von entsprechenden (Vor-)Kenntnissen in DL4J

Erkennungsrate	1: 66,50% 2: 65,95% 3: 56,69% Durchschnitt: 63,05%	1: 94,94% 2: 94,94% 3: 94,94% Durchschnitt: 94,94% ⁵
----------------	---	--

Tabelle 1 Vergleich Eigenimplementierung mit DL4J

Die unterschiedlichen Laufzeiten lassen sich noch durch einen nicht-optimierten Programmcode der Eigenimplementierung (zumindest ansatzweise) erklären, die deutliche Diskrepanz in der Erkennungsrate jedoch nicht ohne tiefergehende Analyse präzisieren - auch wenn ein Code-Fehler in der Eigenentwicklung wahrscheinlich scheint.

Somit unterstreichen die bestimmten Kennziffern neben den bereits in Kapitel 5.3.4 herausgearbeiteten Problemen die Entscheidung, auf das DL4J-Framework zu setzen und weisen auf (messbare) Schwierigkeiten hin, ein Neuronales Netz effizient von Grund auf zu entwickeln.

6.2 Vergleich klassisches FFN mit CNN

Wie in Kapitel 3.3.2.3 erwähnt gelten CNNs in der Praxis bei Bildanalysen als tendenziell effizienter im Vergleich zu klassischen, voll-verknüpften FFNs. Dies soll nun auch experimentell, auf Basis von Deeplearning4j und unter Verwendung des SD-19-Datensatzes gezeigt werden.

Gleichzeitig soll auch jenes KNN bestimmt werden, welches für die Praxistests im nächsten Abschnitt herangezogen werden sollte.

6.2.1 Hyperparameter

Hierzu wird einerseits in der Klasse Sd19MLP ein voll-verknüpftes, 3-schichtiges FFN (mit 400 Neuronen in der verdeckten Schicht, s. [Liu14]) sowie ein CNN in der Klasse Sd19CNN definiert. Beide Klassen finden sich im Projekt d14j.

Das klassische FFN wird in 2 Varianten erprobt: einmal ohne und einmal mit einem Dropout von 50% in der verdeckten Schicht. Die restlichen Parameter stimmen ansonsten überein.

⁵ Die ermittelte Erkennungsrate ist zwar nicht schlecht, jedoch mit Respektabstand zu den besten Ergebnissen, welche bei 99,79% liegen (die jedoch mit hochspezialisierten Netzen erreicht wurden, vgl. [Wik-e]).

Das CNN orientiert sich grundsätzlich am entsprechenden Code aus Kapitel 5.4.2.2 (d.h. 2 Paare Convolutional-/Subsampling-Layer), kommt jedoch gleichfalls in 2 Konfigurationen zum Einsatz: jeweils vor der Ausgabeschicht einmal mit 2 voll-verknüpften Schichten zu je 200 bzw. 100 Neuronen sowie eine vollverknüpfte Schicht mit 500 Neuronen und einem Dropout von 50%.

Beide KNNs besitzen eine Eingangs- bzw. Ausgangsschicht, bestehend aus 784 bzw. 62 Neuronen. Ebenso wurde jeweils ein Momentum mit Wert 0,9 festgelegt, sowie eine Xavier-Gewichts-Initialisierung bzw. eine L2-Regularisierung (mit Wert 0,0005) definiert. Zur Fehlerbestimmung wird die negative Log-Likelihood-Funktion und zur Neuronen-Aktivierung die Softmax-Funktion herangezogen.

Unterschiede finden sich in der Batchgröße (64 für das voll-verknüpfte FFN, 32 für das CNN) sowie bei der Lernrate: während das FFN eine konstante Größe von 0,05 definiert, verwendet das CNN eine variable Lernrate.

Als Epochenzahl wird 20, 40 und (exklusiv für das FFN) 100 festgelegt.

Die gewählten Optionen stellen natürlich nur eine (begrenzte) Auswahl möglicher KNN-Konfigurationen dar, sollten jedoch dennoch einige Trends bzw. Heuristiken bestimmbar machen.

6.2.2 Trainingsdaten

Um nicht Gefahr zu laufen, durch Einsatz aller NIST SD-19 Datensätze die Trainings-Laufzeit exorbitant zu erhöhen, wird eine Reduktion durch die zufällige Wahl von nur (maximal) 2000 bzw. 4000 Trainings-Daten pro zu lernendes Zeichen (und damit, durchschnittlich, um den Faktor 6 bzw. 3) sowie analog von 1000 Test-Daten je Zeichen vorgenommen.

Dies geschieht in der Klasse `DataRandomSplit` (Projekt `d14j`), welche, eigenständig lauffähig, über die in Kapitel 5.1.3.2 vorgestellte Ordner-Struktur iteriert und die gewünschte Anzahl an zufällig gewählten Bildern in einen neuen Ordner kopiert.

Zwei unterschiedliche Werte für die Trainingsbilder-Anzahl werden deshalb gewählt, um den Einfluss auf Laufzeit und Erkennungsrate bestimmen zu können – auch wenn beim direkten Vergleich bedacht werden sollte, dass aufgrund der gewählten Strategie (zufälligen Wahl der einzelnen Bilder) wahrscheinlich (wenn überhaupt) nur eine partielle Überschneidungen der Bildmengen vorliegt und das Sample mit 4000 Bildern/Zeichen nur statistisch eine höhere Varianz aufweisen sollte.

6.2.3 Ergebnis

Ein Vergleich ist sowohl horizontal als auch vertikal möglich: zum einen können verschiedene Konfigurationen eines klassischen FFN bzw. CNN verglichen, zum anderen beide KNN-Ausprägungen unter ähnlichen, über Trainingsbilder- und Epochen-Anzahl gesteuerten Rahmenbedingungen gegenübergestellt werden.

	Klassisches FFN	CNN
Laufzeit (2000 Bilder/Zeichen, ohne Dropout)	20 Epochen: 18min, 22s 40 Epochen: 27min, 45s 100 Epochen: 73min, 12 s	20 Epochen: 67min, 56s 40 Epochen: 138min, 33s
Erkennungsrate	20 Epochen: 69,67% 40 Epochen: 70,83% 100 Epochen: 70,96%	20 Epochen: 74,62% 40 Epochen: 73,74%
Laufzeit (2000 Bilder/Zeichen, mit Dropout)	20 Epochen: 15min, 9s	40 Epochen: 143min, 7s
Erkennungsrate	20 Epochen: 64,95%	40 Epochen: 72,43%
Laufzeit (4000 Bilder/Zeichen, ohne Dropout)	20 Epochen: 32min, 28s 40 Epochen: 49min, 20s 100 Epochen: 136min, 49s	20 Epochen: 130min, 59s
Erkennungsrate	20 Epochen: 70,77% 40 Epochen: 71,60% 100 Epochen: 72,32%	20 Epochen: 73,29%
Laufzeit (4000 Bilder/Zeichen, mit Dropout)	20 Epochen: 29min, 52s	40 Epochen: 241min, 53s
Erkennungsrate	20 Epochen: 65,76%	40 Epochen: 72,97%

Tabelle 2 Vergleich klassisches FFN mit CNN

Auch wenn nicht alle Variationen möglicher (Hyper-)Parameter-Werte probiert wurden und auch, auf Basis der geringen Varianz der DL4J-Messungen in Kapitel 6.1.3, nur eine

Messung pro Konfiguration gemacht wurde (was zusätzlich einiges an Zeit erspart und den zeitlichen Rahmen der Diplomarbeit nicht zu sehr strapaziert), sticht manches ins Auge:

- Das CNN schneidet, ganz wie erwartet und bezogen auf die Erkennungsrate, generell besser als das voll-verknüpfte, klassische FFN ab. Zudem ist (als kleines Detail) die Export-Größe eines trainierten CNN geringer als jene für ein klassisches FFN.
- Auch verspricht das CNN eine höhere Erkennungsrate bei ähnlicher Trainingszeit (das klassische FFN trainiert zwar schneller bei gleich gewählter Trainingsbilder- und Epochenzahl, liefert dann jedoch eine schlechtere Test-Performance).
- Eine Erhöhung der Anzahl an Trainingsbildern führt zwar zu einer längeren Trainingszeit (wenn auch ein nicht-linearer Zusammenhang vorzuliegen scheint), ermöglicht jedoch i.A. eine bessere Erkennungsrate.
- Gleichsam lässt sich (zumindest beim klassischen FFN) ein positiver Einfluss auf die Erkennungsrate bei Erhöhung der Epochenanzahl festmachen - wenn dies auch wiederum durch eine höhere Trainingszeit erkaufte wird.
- Der Einsatz von Dropout führt, zumindest in der gewählten Variante, zwar zu einer Verringerung der Laufzeit, gleichzeitig aber auch zu einem (überproportionalen) Einbruch in der Erkennungsrate. Es scheint, dass ein blinder/unüberlegter Einsatz dieser Strategie sogar mehr schadet als nützt.

Natürlich sind die beschriebenen Erkenntnisse mit Vorsicht zu genießen und bedürften, zur Festigung, weitere (zeitintensive) Untersuchungen⁶.

6.3 Erkennen von Handschrift

Die zuvor ermittelten Ergebnisse sind zufriedenstellend genug⁷ um bspw. auf Basis des bei 2000 Bildern/Zeichen und 20 Epochen trainierten CNN (jenem mit der höchsten Erkennungsrate) die Praxistauglichkeit zu erproben.

6.3.1 Beispiel

Die zu analysierende IBAN wird, gemäß Vorgaben aus Kapitel 5.2.1, händisch erfasst und anschließend per Scanner digitalisiert:

⁶ Eventuell würde sich dadurch auch eine Erklärung für die (Trend-)Ausreißer bei der CNN-Performance (niedere Erkennungsrate trotz höherer Epochen- bzw. Trainingsbilder-Anzahl) finden.

⁷ Eine Erkennungsrate von knapp 75% erscheint zwar im ersten Moment geringer als erwünscht, entspricht aber bspw. bei einem Pool aus 62 möglichen Ergebnissen einer um den Faktor 46,5 erhöhten Wahrscheinlichkeit als simples Raten und wäre lt. [Liu14] ein durchschnittliches Ergebnis.

A photograph of a handwritten IBAN 'AT022050302101023600' in dark ink on a light background.

Abbildung 29 Beispiel einer händisch erfassten IBAN

Die Bildvorverarbeitung extrahiert anschließend die einzelnen Zeichen:

The same handwritten IBAN 'AT022050302101023600' where each character is enclosed in a small black rectangular box, representing the result of image segmentation.

Abbildung 30 Segmentierung einer IBAN

Diese werden nun durch das CNN ausgewertet. Ohne Beachtung der IBAN-Syntax ergibt sich folgende Konsolen-Ausgabe:

```
[main] INFO com.depies.dl4j.Sd19CNN - *** OUTPUT ***  
[main] INFO com.depies.dl4j.Sd19CNN - AT02205030L1oiozg600
```

Listing 36 Auswertung einer händisch erfassten IBAN ohne Syntax-Beachtung

Dieses Ergebnis ist, wenn auch Stellen bereits übereinstimmen, nicht wirklich brauchbar, weshalb, Kapitel 5.5 folgend, eine weitere Analyse inklusive IBAN-Interpretation unternommen wird. Diese ist erfolgreich:

```
[main] INFO com.depies.dl4j.Sd19CNN - *** OUTPUT ***  
[main] INFO com.depies.dl4j.Sd19CNN - AT02 2050 3021 0102 3600  
[main] INFO com.depies.dl4j.Sd19CNN - valid: true
```

Listing 37 Ergebnis einer erfolgreichen IBAN-Identifikation

6.3.2 Probleme (und mögliche Lösungsansätze)

Das oben gezeigte Beispiel ist erfreulich und demonstriert, dass die implementierte Software sowie das trainierte Netzwerk grundsätzlich in der Lage sind, eine Handschrift erkennen bzw. auswerten zu können, dennoch lassen sich bei Anwendung verschiedener Praxisbeispiele eine Reihe wiederkehrender Probleme feststellen.

6.3.2.1 Bildvorbereitung

Die korrekte Extraktion einzelner Zeichen hängt stark von einer durchgehenden Schriftführung und entsprechenden –dicke ab. Dies lässt sich bei folgendem Bild demonstrieren:

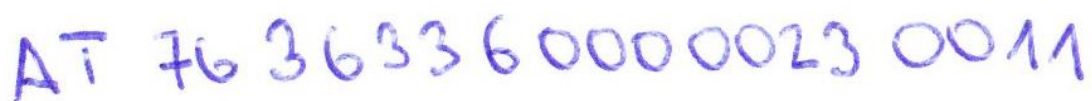
A photograph of a handwritten IBAN 'AT 76 3633 6000 0023 0011' in blue ink. The characters are more spread out and less uniform in thickness compared to the previous example.

Abbildung 31 Händisch schlecht erfasste IBAN

Die Anwendung der Zeichen-Extraktion mit den festgelegten Parametern führt zu einem nicht weiter verwertbaren Ergebnis, da nur die wenigsten Zeichen korrekt erkannt werden:



Abbildung 32 Fehlerhafte Segmentierung einer händisch schlecht erfassten IBAN

Die besten Ergebnisse wurden deshalb auch bei Verwendung von Filzstiften erzielt.

6.3.2.2 Trainingsdaten

Die zufällige Auswahl von maximal nur 4000 Trainingsbildern je erlernendem Zeichen (bei einem Pool von z.T. über 38.000 Bildern) geht natürlich auf Kosten möglicher Variabilität und erschwert zudem die Reproduzierbarkeit von Trainingsergebnissen im Falle einer neuen Wahl von Trainingsbildern.

Auch die Tatsache, dass NIST SD-19 auf in den USA ermittelten Daten beruht, nimmt Einfluss auf die globale Verwertbarkeit darauf trainierter KNNs. Beispielsweise wird die Ziffer 1 meist ohne dem im deutschsprachigen Raum üblichen diagonalen oberen Strich geschrieben (und ähnelt daher mehr der römischen Ziffer I bzw. dem Buchstaben I):

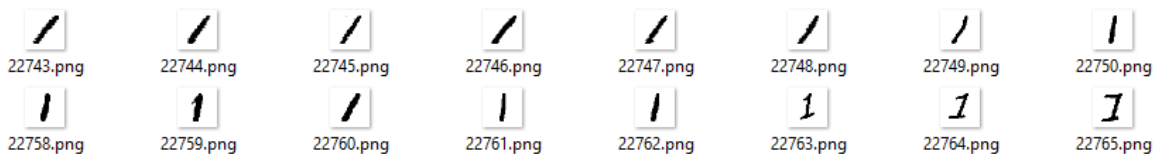


Abbildung 33 Beispiele für die Ziffer 1 aus MNIST SD-19

Die Ziffer 7 hingegen ähnelt oft der bei uns üblichen Ziffer 1:

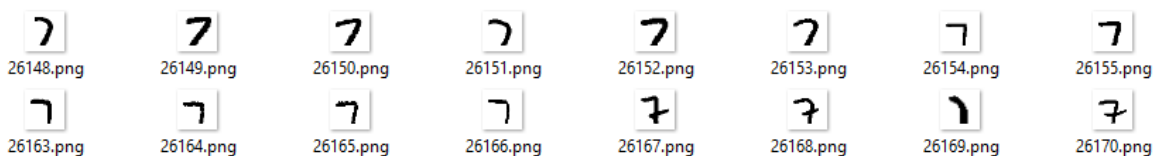


Abbildung 34 Beispiele für die Ziffer 7 aus MNIST SD-19

Eine Verwechslung von Ziffer und Buchstabe (1 und I, 0 und O) lässt sich noch über den Kontext abfangen (siehe Kapitel 5.5.3), der Ähnlichkeit zwischen den beiden Ziffern 1 und 7 vorzubeugen ist schon mit deutlich mehr Unsicherheit verbunden.

Zum einen kann einfach ein neues, zufälliges Set von Trainingsbildern bestimmt werden, welches möglicherweise ein günstigeres Verhältnis verschiedener Varianten der Ziffer 1

und 7 aufweist. Dies kann natürlich auch durch die manuelle Auswahl entsprechender Trainingsbildern erreicht werden.

Eine weitere (wenn auch nicht implementierte) Möglichkeit besteht darin, bei einer nicht validierbaren IBAN Vorkommnisse der Ziffer 1 mit 7 (und umgekehrt) schrittweise zu tauschen und jeweils anschließend eine neuerliche Validierung anzustoßen. Die Gefahr, dass dieser Ansatz das tatsächliche Ergebnis verfälscht, fällt umso geringer aus, je kleiner die Zahl erkannter Ziffern 1 bzw. 7 ist.

6.3.2.3 Bildinterpretation

Ein Beispiel der Ziffer 2 als 5. Stelle einer IBAN:



Abbildung 35 Beispiel der 5. Stelle einer IBAN

Das KNN ermittelt folgende Wahrscheinlichkeiten der Zugehörigkeit zu einer bestimmten Ergebnisklasse (ohne Beachtung der Groß- und Kleinschreibung):

V: 0.7493, U: 0.1562, N: 0.0423, X: 0.0107, D: 0.0057, L: 0.0039, F: 0.0037, O: 0.0037, J: 0.0036, W: 0.0033, K: 0.0032, T: 0.0027, I: 0.0024, B: 0.0015, Y: 9.0E-4, H: 8.0E-4, 1: 8.0E-4, R: 7.0E-4, S: 7.0E-4, 2: 7.0E-4, P: 6.0E-4, 0: 6.0E-4, 5: 0.0588, 8: 0.0019, Z: 5.0E-4, A: 5.0E-4, C: 2.0E-4, M: 2.0E-4, 4: 2.0E-4, 6: 0.0094, Q: 1.0E-4, G: 1.0E-4, E: 1.0E-4

Listing 38 Beispiel eines Zeichen-Interpretationsergebnisses

Aus dem Kontext ist bekannt, dass es sich um eine Ziffer handeln muss, weshalb nicht weniger als die 16 wahrscheinlichsten, ursprünglich bestimmten Zuordnungen ignoriert werden und deshalb die Darstellung, fälschlicherweise, als 1 und nicht als 2 interpretiert wird. Die IBAN-Validierung schlägt somit aufgrund einer einzigen, absoluten Wahrscheinlichkeitsabweichung von 0,01% fehl.

In Abwandlung der zuvor vorgestellten Präventionsstrategie zur Vermeidung von Verwechslungen der Ziffer 1 und 7 könnte allgemein der Ansatz gewählt werden, bei IBAN-Validierungsfehlern den (schrittweise erfolgenden) Austausch einzelner Zeichen mit dem nächst-wahrscheinlichen Ergebnis durchzuführen.

Auch ohne Kontext-Beachtung könnte dadurch zumindest eine Art Vertrauens-Maß definiert werden, indem das wahrscheinlichste Ergebnis einen definierten (relativen oder absoluten) Mindest-Abstand zur nächst-wahrscheinlichen Interpretation haben muss (und somit statistische Ungenauigkeiten bzw. Rauschen besser in den Griff zu bekommen sind). Wäre diese Bedingung bspw. für nur ein Zeichen verletzt, würde das gesamte Analyse-Ergebnis in Zweifel gezogen (und entsprechend bewertet) werden.

6.4 Erkennen von Maschinschrift

Nachdem auf Basis von SD-19 exklusiv das Erlernen von Handschriften trainiert wird, stellt sich die Frage, wie das Ergebnis Analyse von Maschinschrift ausfällt – immerhin handelt es sich dabei um ein Schriftbild, welches zwar viel Ähnlichkeit mit einer Handschrift aufweisen kann, dennoch keinem Trainingsschritt als Vorlage diene.

Grundlage des folgenden Tests bildet der Scan eines IBAN-Aufdrucks auf einem Zehlschein:

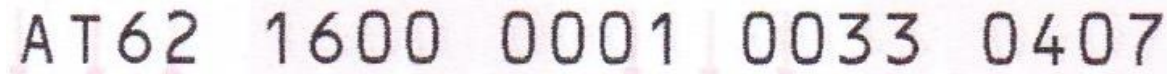


Abbildung 36 IBAN in Maschinschrift

Die Zeichen-Extraktion gelingt aufgrund der nicht-proportionalen Schriftart (und gleichmäßigen Ausrichtung) problemlos:



Abbildung 37 Zeichenextraktion bei IBAN in Maschinschrift

Die Interpretation durch ein CNN, welches die handschriftlich erfasste IBAN aus Kapitel 6.3.1 noch problemlos erkennen konnte, schlägt jedoch fehl, die ermittelten Werte entsprechen nicht der Vorlage und damit einer (gültigen) IBAN:

```
[main] INFO com.depies.dl4j.Sd19CNN - *** OUTPUT ***
[main] INFO com.depies.dl4j.Sd19CNN - AI62 1600 0007 0033 0607
[main] INFO com.depies.dl4j.Sd19CNN - invalid IBAN: AI621600000700330607
```

Listing 39 Fehlgeschlagene IBAN-Interpretation

6.4.1 Analyse

Eine nähere Analyse der Abweichungen liefert folgende Erkenntnisse:

- Ohne Beachtung der Groß-/Kleinschreibung wäre die 2. Stelle korrekt als T klassifiziert worden (Wahrscheinlichkeit: 32,26%), einzig die kumulierte Wahrscheinlichkeit für I (18,87% + 17,66%) lag über jener für T (34,79%).
- Für die 12. Stelle wurde eine Wahrscheinlichkeit von 3,42% für 1 sowie 4,45% für 7 bestimmt. Im Vergleich dazu betrug die Wahrscheinlichkeit für 1 beim 5. Zeichen 8,03% und für 7 1,43%.
- Die 18. Stelle wurde mit einer Wahrscheinlichkeit von 52,78% als 6 und nur 0,32% als 4 interpretiert.

Beachtenswert dabei ist, dass keiner dieser Fehler auf die Tatsache zurückzuführen ist, dass eine Maschinschrift untersucht wurde.

Entsprechend sind (mögliche) Lösungsansätze so formulierbar, dass auch deren reine Anwendung im Kontext von Handschrift-Analysen eine Verbesserung darstellt:

- Nachdem es wahrscheinlich ist, dass eine händisch erfasste IBAN immer mit Großbuchstaben geschrieben wird, könnte auch allein die Wahrscheinlichkeit der einzelnen Großbuchstaben für die Interpretation herangezogen werden.
- Der Verwechslung von 1 und 7 könnte durch das zuvor vorgestellte Vertrauens-Maß vorgebeugt werden: lt. Analyse ist die 7 nur um den Faktor 1,3 wahrscheinlicher als die 1, was vielleicht gerade bei diesen 2 Ziffern ein zu geringer Abstand ist.
- Die Verwechslung von 4 und 6 deutet hingegen auf eine generelle Schwäche des trainierten Netzwerks hin und ließe sich möglicherweise durch ein neuerliches Lernen mit umfangreicheren Trainingsdaten bzw. adaptierten Hyperparametern beheben.

7 Zusammenfassung

7.1 Bewertung

Ausgangspunkt der Arbeit war der Wunsch, sich als kompletter Neuling in die Themenkomplexe Neuronale Netze und Bildanalyse einzuarbeiten, dies entsprechend zu dokumentieren und darauf bauend die praktische Realisierbarkeit einer Schrifterkennungs-Lösung am Beispiel handschriftlich erfasster IBANs zu demonstrieren.

Wenn dabei auch ein lösungsorientierter Ansatz verfolgt wurde, d.h. die Auswahl der präsentierten, theoretischen Inhalte in Kapitel 2 und 3 starken Bezug auf die spätere Umsetzung in Kapitel 5 nahmen, konnten die gesteckten Ziele zu einem großen Teil im geforderten Umfang erreicht werden:

- Wichtige Konzepte aus der Bildbearbeitung und der Theorie hinter Neuronalen Netzen wurden präsentiert.
- Die Zeichen-Extraktion aus Bildern war, wenn auch unter Auflagen (s. Kapitel 5.2.1), möglich.
- Verschiedene, ausgewählte KNN-Varianten sowie entsprechende Implementierungs-Strategien (komplette Eigenentwicklung bzw. Integration eines fertigen Frameworks) konnten erprobt und verglichen werden. Dabei wurde festgestellt, dass eine Eigenentwicklung zwar grundsätzlich möglich ist, jedoch in nahezu allen Belangen einer etablierten Lösung hinterherhinkt.
- Die verschiedenen Implementierungs-Teile (Zeichen-Extraktion, KNN-Training und -Auswertung, Interpretation) konnten soweit abgestimmt werden, dass der vollständige Durchlauf von der Bild-Vorlage einer IBAN bis hin zur maschinellen Interpretation (wenn auch nicht vollständig automatisiert, sondern von einem notwendigen, manuellen Eingreifen zwischen den einzelnen Schritten unterbrochen) möglich war.

Die Arbeit bildet damit eine gute Basis für weitergehende Untersuchungen.

7.2 Erkenntnisse

Eine entscheidende Schlussfolgerung aus Kapitel 6 ist die Feststellung, dass das Wissen zum Kontext der zu untersuchenden Daten massiven Einfluss auf die ermittelte Performance (und damit unmittelbar auch auf die Praxistauglichkeit) von KNNs nimmt.

Ein solches (a-priori) Wissen ist dabei nicht nur für die Trainingsphase (durch Auswahl passender Trainingsdaten) und für die Wahl der Hyperparameter eines KNN entschei-

dend, sondern mindestens gleich wichtig für die nachfolgende Interpretation der Klassifizierungs-Ergebnisse: ist bspw. genau spezifiziert, welche Syntax bei einem zu analysierenden Text vorherrscht und fließt dieses Wissen in die Zeichen-Analyse mit ein, kann das Mess-Ergebnis (zum Positiven) kippen (vgl. Kapitel 6.3).

Dies ist umso wichtiger, da bei technischen Definitionen (wie eben einer IBAN) im Gegensatz zum normalen Alltags-Text (vgl. [Pre] oder [Tel03]) Redundanz i.d.R. weitestgehend vermieden wird bzw. meist nur eine Fehlererkennung (wie die Prüfsumme einer IBAN) zur Verfügung steht. Dafür ist meist ein genauer Rahmen spezifiziert (Syntax, Pattern), der den Kontext und damit die Gültigkeit eines entsprechenden Terms vorgibt.

Die Bedeutung des Kontexts lässt sich auch anhand folgender Überlegung demonstrieren:

Die höchste, ermittelte Erkennungsrate eines CNN aus Kapitel 6.2.3 betrug knapp 75%. Dennoch war es möglich, eine 20-stellige IBAN korrekt zu interpretieren, was (grob geschätzt) einer Wahrscheinlichkeit von $0,75^{20} = 0,003$, d.h. 0,3% entsprechen würde. Auch wenn dieser Wert zu nieder angesetzt sein sollte (u.a. fallen Verwechslungen bei Groß- und Kleinschreibung nicht ins Gewicht), zeigt sich dennoch (vgl. Listing 36 und Listing 37), dass überhaupt erst die Interpretation auf Basis eben des Kontexts die Chance auf ein verwertbares Klassifizierungs-Ergebnis liefert.

7.3 Ausblick

Wenn auch viele Themen nur angeschnitten wurden, konnten, neben den bereits erwähnten (vgl. Kapitel 6.3.2 und Kapitel 6.4.1), einige relevante Punkte bzgl. Optimierung und Ausbau herausgearbeitet werden:

Zum einen ist die gewählte Umsetzung der Bildvorbearbeitung recht starr und garantiert nur unter nahezu idealen Bedingungen verwertbare Segmentierungen. Eine Anpassung der aktuellen Parameter-Werte könnte, auf Basis entsprechender Messreihen, bereits eine Besserung bringen, auch die Implementierung alternativer Zeichen-Detektions-Strategien (vgl. [Par11], 321 ff.) wäre denkbar. Eine ebenso interessante Methode wäre es, die entsprechenden Werte variabel zu setzen und bspw. die anschließend bestimmte KNN-Erkennungsrate als Feedback zur Qualitätsbestimmung einzusetzen (vorausgesetzt der Wert des zu extrahierenden Zeichens ist vorab bekannt).

Auf Basis der bisher gesammelten Erfahrungen könnte zudem folgende, spezifische Strategie zur (österreichischen) IBAN-Erkennung umgesetzt werden: anstelle eines einzigen KNN kommen zwei KNNs zum Einsatz – eines spezialisiert auf Ziffern, das andere auf 52 Buchstaben (Groß- und Kleinschreibung der Buchstaben A-Z). In Summe dürfte sich ein deutlicher Anstieg in der Performance bemerkbar machen, da für jede Stelle einer IBAN klar definiert wäre, welches der beiden KNNs zum Einsatz kommt.

Die Auswirkung von größeren Eingangsdaten (d.h. Bilder $> 28 \times 28$ Pixel) auf die Performance (Erkennungsrate, Laufzeit) insbesondere von CNNs wurde noch nicht untersucht: einerseits versprechen höhere Auflösungen mehr Details und damit auch eine genauere Klassifizierung, andererseits bedingen mehr Eingaben/Verknüpfungen eine höhere Zahl an Trainingsdaten – und damit insgesamt eine (deutlich) erhöhte Trainingszeit.

Gleiches gilt für (ultra-)lange Trainingszeiten: subjektiv gesehen mag eine bisherige, maximale Dauer von ca. 4 Stunden lang sein (nicht zuletzt auch deshalb, da die eingesetzte Hardware auf Hochtouren lief), trotzdem wird innerhalb dieses Zeitrahmens vermutlich nicht das Maximum aus einem KNN geholt werden können – interessant wäre deshalb, ab welcher Anzahl von Trainingsdaten bzw. Epochen (und damit auch ab welcher Dauer) eine entsprechende Sättigung stattfindet.

Literaturverzeichnis

Die gelisteten URLs wurden zuletzt am 28.9.2017 abgerufen.

- [Ben12] Y. Bengio: Practical recommendations for gradient-based training of deep architectures, <https://arxiv.org/abs/1206.5533>
- [Bis06] C. Bishop: Pattern Recognition and Machine Learning, New York, Springer, 2006
- [Cir] D. Ciregan et al.: Multi-column Deep Neural Networks for Image Classification, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6248110>
- [ct16] c't: Netzgespinste, <https://www.heise.de/ct/ausgabe/2016-6-Die-Mathematik-neuronaler-Netze-einfache-Mechanismen-komplexe-Konstruktion-3120565.html>
- [Geg] K. Gegenfurter et al.: Visuelle Informationsverarbeitung im Gehirn, <http://www.allpsych.uni-giessen.de/karl/teach/aka.htm>
- [Gro] J. Groß: Wer, Wie, Was: Die Verarbeitung von visuellen Informationen, <https://www.dasgehirn.info/wahrnehmen/sehen/wer-wie-was-die-verarbeitung-von-visuellen-informationen>
- [Hay99] S. Haykin: Neural Networks – A Comprehensive Foundation, Upper Saddle River, Pearson Education, 1999
- [Hei16] Heise Online: Barrierefreiheit: Facebook lässt Bilder beschreiben, <https://www.heise.de/newsticker/meldung/Barrierefreiheit-Facebook-laesst-KI-Bilder-beschreiben-3162331.html>

- [Hei17] Heise Online: Machine Learning: Facebook erweitert Einsatz von Caffe2, <https://www.heise.de/developer/meldung/Machine-Learning-Facebook-erweitert-Einsatz-von-Caffe2-3792904.html>
- [Her12] S. Herculano-Houzel: The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost, <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3386878/>
- [Hin] G. Hinton et al.: A fast learning algorithm for deep belief nets, <http://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>
- [Hoc] S. Hochreiter, J. Schmidhuber: Long short-term Memory, <http://www.bioinf.jku.at/publications/older/2604.pdf>
- [Idx] IDX file format, http://www.fon.hum.uva.nl/praat/manual/IDX_file_format.html
- [Jon] A. Jones: An Explanation of Xavier initialization, <http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization>
- [Kar] A. Karpathy: Convolutional Neural Networks for Visual Recognition, <http://cs231n.github.io/convolutional-networks/>
- [KrD] D. Kriesel: Neuronale Netze, http://www.dkriesel.com/en/science/neural_networks
- [Kri] A. Krizhevsky et al.: ImageNet Classification with Deep Convolutional Neural Networks, <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- [LeC] Y. LeCun: LeNet-5, convolutional neural networks,

- <http://yann.lecun.com/exdb/lenet/>
- [Liu14] J. Liu: Neural Networks in the Wild: Handwriting Recognition, <https://de.slideshare.net/guard0g/neural-networks-in-the-wild-handwriting-recognition>
- [Mar] D. Marr: Vision. A Computational Investigation into the Human Representation and Processing of Visual Information, Cambridge, The MIT Press, 2010
- [May] M. Matthew: Top 10 Deep Learning Projects on Github, <http://www.kdnuggets.com/2016/01/top-10-deep-learning-github.html>
- [MPI] Max-Planck Institut für intelligente Systeme: Autonomes Maschinelles sehen, <http://www.is.mpg.de/de/avg>
- [Nie] M. Nielsen: Neural Networks and Deep Learning, <http://neuralnetworksanddeeplearning.com/>
- [NVI] NVIDIA: What is GPU-accelerated computing, <http://www.nvidia.com/object/what-is-gpu-computing.html>
- [OCV-a] OpenCV: Smoothing images, http://docs.opencv.org/2.4/doc/tutorials/imgproc/gaussian_median_blur_bilateral_filter/gaussian_median_blur_bilateral_filter.html
- [OCV-b] OpenCV: Contour Features, http://docs.opencv.org/trunk/dd/d49/tutorial_py_contour_features.html
- [OCV-c] OpenCV: Contours Hierachy, http://docs.opencv.org/trunk/d9/d8b/tutorial_py_contours_hierarch

y.html

- [Pap] S. Papert: The Summer Vision Project,
<https://dspace.mit.edu/handle/1721.1/6125>
- [Par11] Parker, J.R.: Algorithms for Image Processing and Computer Vision, Second Edition. Indianapolis, Wiley Publishing, Inc., 2011
- [Pow] V. Powell: Image Kernels explained visually,
<http://setosa.io/ev/image-kernels/>
- [Pre] E. Preussler: Redundanz – aber richtig, <http://www-stud.rbi.informatik.uni-frankfurt.de/~erps/redundanz.html>
- [Quo] Quora: What is the implication of the Universal Approximation Theorem over deep learning methodology,
<https://www.quora.com/What-is-the-implication-of-the-Universal-Approximation-Theorem-over-deep-learning-methodology>
- [Rei10] G. Reif, Moderne Aspekte der Wissenschaftsvermittlung, Diplomarbeit, TU Graz, 2000
- [Ros] F. Rosenblatt: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.588.3775>
- [Rum] D. Rumelhart et al.: Learning representations by back-propagation errors,
https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

- [Sri14] N. Srivastava et al.: Dropout: A Simple Way to Prevent Neural Networks from Overfitting,
<https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>
- [Std15] Der Standard Online: TensorFlow: Googles Maschinenlern-Software ist nun Open Source,
<http://derstandard.at/2000025375689/TensorFlow-Googles-Maschinenlern-Software-ist-nun-Open-Source>
- [StE-a] StackExchange: How to choose the number of hidden layers and nodes in a feedforward neural network,
<https://stats.stackexchange.com/q/181>
- [StE-b] StackExchange: Does Dimensionality curse effect some models more than others, <https://stats.stackexchange.com/q/186184>
- [StE-c] StackExchange: Danger of setting all initial weights to zero in Backpropagation, <https://stats.stackexchange.com/q/27112>
- [StE-d] StackExchange: Cross-Entropy or Log Likelihood in Output Layer, <https://stats.stackexchange.com/q/198038>
- [StO-a] Stackoverflow: How does the back-propagation algorithm deal with non-differentiable activation functions,
<https://stackoverflow.com/q/30236856>
- [StO-b] Stackoverflow: Why must a nonlinear activation function be used in a backpropagation neural network,
<https://stackoverflow.com/q/9782071>
- [StO-c] Stackoverflow: Performance of memory consumption difference between Array and ArrayList,
<https://stackoverflow.com/q/27625161>

- [Tel03] Telepolis: Unlguailbch,
<https://www.heise.de/tp/features/Unlguailbch-3431355.html>
- [TR09] Technology Review: Digitale Bilderkennung,
<https://www.heise.de/tr/artikel/Digitale-Bilderkennung-833604.html>
- [Wal] A. Wallner: Neuronale Netze, http://www.mathematik.uni-ulm.de/stochastik/lehre/ss07/seminar_sl/ausarbeitung_wallner.pdf
- [Wik-a] Wikipedia: Gehirn – Rechenleistung und Leistungsaufnahme,
https://de.wikipedia.org/wiki/Gehirn#Rechenleistung_und_Listungsaufnahme
- [Wik-b] Wikipedia: Douglas-Peucker-Algorithmus,
<https://de.wikipedia.org/wiki/Douglas-Peucker-Algorithmus>
- [Wik-c] Wikipedia: Rezeptives Feld,
https://de.wikipedia.org/wiki/Rezeptives_Feld
- [Wik-d] Wikipedia: Comparison of deep learning software,
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software
- [Wik-e] Wikipedia: MNIST database,
https://en.wikipedia.org/wiki/MNIST_database
- [Wik-f] Wikipedia: Texterkennungssysteme,
<https://de.wikipedia.org/wiki/Texterkennung>

Hersteller- und Softwareverzeichnis

[BLA]	Basic Linear Algebra Subprograms, http://www.netlib.org/blas/
[DL4]	Deep Learning for Java, https://deeplearning4j.org/
[Ecl]	Eclipse, http://www.eclipse.org/
[Git-a]	Preprocess NIST SD19, https://github.com/ProjectAGI/Preprocess_NIST_SD19
[Git-b]	java-iban, https://github.com/barend/java-iban
[Jav]	Java, http://www.oracle.com/technetwork/java/
[MNI]	MNIST databse, http://yann.lecun.com/exdb/mnist/
[ND4]	N-Dimensional Arrays for Java, http://nd4j.org/
[NIS]	NIST Special Database 19, https://www.nist.gov/srd/nist-special-database-19
[OCV]	OpenCV, http://www.opencv.org/
[Sky]	SkyMind, https://skymind.ai/

[SLF] Simple Logging Facade For Java, <https://www.slf4j.org/>

Anhang

Setup	A-I
Historischer Abriss	A-V
Erkennen von allgemeinem Text.....	A-VII
Inhalt der CD.....	A-IX

Setup

java-iban

Die Integration von java-iban gestaltet sich dank Maven-Unterstützung denkbar einfach. Folgende Zeilen sind hierfür im entsprechenden pom.xml einzutragen:

```
<dependency>
  <groupId>n1.garvelink.oss</groupId>
  <artifactId>iban</artifactId>
  <version>1.5.0</version>
</dependency>
```

Listing 40 java-iban Maven-Dependency

OpenCV

OpenCV lässt sich wie folgt für die Java-Entwicklung unter Eclipse einbinden:

Die aktuelle Version von OpenCV (für die vorliegende Arbeit wurde noch Version 3.1.0 verwendet, aktuell wäre 3.2.0) muss heruntergeladen und anschließend in einem beliebigen Verzeichnis entpackt werden. Die entpackte Ordnerstruktur weist u.a. die Hierarchie build/java auf. In diesem Verzeichnis findet sich die JAR-Datei opencv-<Version>, zudem, getrennt nach 32- oder 64-Bit-Plattform, jene DLL-Dateien, welche für das Java-Binding der OpenCV-Funktionen sorgen.

In Eclipse wird diese JAR-Datei als neue User Library (unter Window > Preferences, anschließend Java > Build Path > User Libraries) ergänzt: Hierfür einfach mit New einen beliebigen Namen setzen und anschließend mit Add External JARs... die JAR-Datei einbinden.

Als nächstes muss die so ergänzte User Library in den Build Path des Java-Projekts übernommen werden: dies lässt sich am schnellsten durch einen Rechtsklick auf den Projekt-Namen im Package Explorer und anschließender Wahl von Build Path > Add Libraries... erreichen. Im aufpoppenden Fenster wird User Libraries gewählt, anschließend kann der zuvor vergebene Name für die User Library selektiert werden.

Final wird die native OpenCV-DLL-Bibliothek mit folgender Zeile im Java-Code geladen:

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

Listing 41 Laden einer nativen DLL in Java zur Laufzeit

Deeplearning4J

DL4J wird als Maven-Plugin zur Verfügung gestellt:

```
<dependency>
  <groupId>org.deeplearning4j</groupId>
  <artifactId>deeplearning4j-core</artifactId>
  <version>0.8.0</version>
</dependency>
```

Listing 42 DL4J Maven-Dependency

Die Integration von ND4J verläuft ähnlich:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-native</artifactId>
  <version>0.8.0</version>
</dependency>
```

Listing 43 ND4J Maven-Dependency

Zusätzlich wird noch SLF4J (Simple Logging Facade for Java, s. [SLF]), eine Logging-Fassade für Java, eingebunden:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.25</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.25</version>
</dependency>
```

Listing 44 SLF4J Maven-Dependencies

Eine GPU-Unterstützung lässt sich bei entsprechend vorhandener Hardware relativ leicht integrieren. Hierfür muss zum einen folgendes Maven-Repository ergänzt werden:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-cuda-8.0</artifactId>
  <version>0.8.0</version>
</dependency>
```

Listing 45 Cuda Maven-Dependency

Zum anderen muss im properties-Bereich des Maven-POM (Project Object Model) die CUDA-Unterstützung aktiviert werden:


```
<nd4j.backend>nd4j-cuda-8.0-platform</nd4j.backend>
```

Listing 46 Aktivierung CUDA-Unterstützung

Ist dieses Property nicht gesetzt bzw. kann trotz Angabe keine CUDA-fähige Hardware gefunden werden, wird implizit das (Default-)CPU-Backend `nd4j-native-platform` verwendet.

Historischer Abriss

Maschinelles Sehen

- Bereits in den 1960er Jahren wurden die ersten Versuche im Bereich maschinellen Sehen unternommen, wobei zuerst noch der Aufwand optimistisch mit einem Sommer-Praktikum geschätzt wurde (vgl. [Pap]).
- In den 70er Jahren wurden Grundlagen bspw. bei der Kantendetektion geschaffen, Analysen beschränkten sich jedoch auf ausgewählte, z.T. synthetische Bilder.
- Die 80er und 90er brachten zwar weitere Algorithmen (z.B. für die Gesichtserkennung), Analyse-Methoden waren jedoch größtenteils auf statistische Methoden (welche entsprechendes Vorwissen bedingen) beschränkt.
- Erst in den 2000er-Jahren waren (katalogisierte) Datenbestände im ausreichenden Maß vorhanden, um (künstliche) neuronale Netze effektiv trainieren zu können und damit brauchbar für Szenen-Analysen zu machen.

Neuronale Netze

Die Entwicklung hin zur aktuellen Theorie (künstlicher) neuronaler Netze geht Hand in Hand mit Fortschritten in den Neuro- und Computerwissenschaften (vgl. [ct16] bzw. [KrD]):

- Die Anfänge liegen in den 40er des vorherigen Jahrhunderts: erste grundlegende Konzepte werden entwickelt, u.a. die Hebb'sche Lernregel, welche besagt, dass die Verbindung zwischen zwei Neuronen verstärkt wird, wenn beide Neuronen gleichzeitig aktiv sind – in einer allgemeinen Form stellt dies die Basis fast aller neuronalen Lernverfahren bis heute dar.
- Die 50er und 60er Jahre gelten als erste Blütezeit neuronaler Forschung: erste künstliche neuronale Netze werden entwickelt u.a. auf Basis des 1958 von Frank Rosenblatt vorgestellten Perzeptron-Modells (s. [Ros]), eine auf Feedback-Mechanismen setzende Frühform lernender Netze. Die Ära findet jedoch ein jähes Ende mit einer Veröffentlichung von u.a. Marvin Minsky (einem Pionier in der Forschung Künstlicher Intelligenz), in welcher im speziellen die Grenzen von Perzeptronen aufgezeigt und allgemein kein gutes Haar an neuronalen Netzen gelassen wurde.
- Dies bewirkte einen Rückgang an Forschungsgeldern, was in den nächsten Jahren zu einer langen Stille in der Forschung führte: diese fand zwar weiterhin statt, größtenteils jedoch ohne Austausch und damit isoliert.

- 1986 wurde zum ersten Mal erfolgreich der Backpropagation-Algorithmus angewandt (s. [Rum]), einem bereits 1974 vorgestellten Verfahren welches die prinzipiellen Schwächen des Perzeptrons umgehen konnte. Minskys Negativabschätzungen waren damit zwar widerlegt, dennoch blieb das Interesse an Neuronalen Netzen enden wollend.
- Jörg Schmidhuber, Wissenschaftlicher Direktor des Schweizer Forschungsinstituts für Künstliche Intelligenz (IDSIA), bezeichnet die 1990er und den Anfang der 2000er Jahre als „Winter der Neuronalen Netze“: die Werkzeuge (Algorithmen) waren zwar vorhanden bzw. wurden auch neue entwickelt (bspw. Convolutional Neural Network), es herrschte jedoch ein Mangel an Trainingsdaten, Rechenleistung und Speicher, weshalb nur wenige komplexe Aufgaben effizient mit künstlichen neuronalen Netzen gelöst werden konnte.
- Die Verfügbarkeit günstiger und leistungsfähiger Hardware in Kombination mit hochwertigen und mit guten Metadaten versehenen Datenmengen, gepaart mit neuen Fortschritten im Bereich Deep Learning (u.a. Deep Belief Networks, s. [Hin]) führte zum Boom Neuronaler Netze wie wir es aktuell erleben – nicht ohne Grund mit den Giganten des Internet (Google, Facebook, Microsoft), den Herren über Daten und Maschinen, an führender Stelle.

Erkennen von allgemeinem Text

Wenn auch nicht eigentlicher Teil der Arbeit (und sich damit im Anhang wiederfindend) drängt sich doch die Frage auf, wie sich ein trainiertes KNN (in Kombination mit dem umgesetzten, IBAN-spezifischen Code) bei der Analyse von händisch erfasstem, allgemeinem Text schlägt (welcher, im Gegensatz zum Beispiel in Kapitel 6.3.1, eben keinen IBAN darstellt).

Hierzu wird ein passender Text digital erfasst:

A photograph of a piece of paper with the handwritten sentence "What does it mean to see" in dark ink. The handwriting is cursive and somewhat informal.

Abbildung 38 Handschriftlich erfasster Text

Die Zeichen-Segmentierung gelingt problemlos, eine Klassifizierung der einzelnen Buchstaben liefert anschließend bspw. folgendes Ergebnis:

```
[main] INFO com.depies.dl4j.Sd19CNN - *** OUTPUT ***  
[main] INFO com.depies.dl4j.Sd19CNN - Whatd0eSIItmaant0See
```

Listing 47 Analyse-Ergebnis eines händisch erfassten Textes

Die Auswertung ist durchaus brauchbar: auch wenn nicht jeder einzelne Buchstabe korrekt interpretiert werden konnte, lässt sich doch verstehen, was geschrieben wurde (dies ist u.a. der Redundanz in natürlichen Sprachen geschuldet, s. Kapitel 7.2).

Dennoch weist das Beispiel auf eine Reihe von Unzulänglichkeiten hin, welche eine entsprechende Adaptierung der entwickelten Klassifizierungs-Umsetzung bedingen würde:

Zum einen erkennt das System keine Leerzeichen (wobei eine entsprechende Anforderung bei IBANs auch nicht gegeben ist). Eine Möglichkeit, Leerzeichen zu erkennen würde jedoch darin liegen die bei der Bildvorverarbeitung bestimmten Koordination und Maße einzelner Zeichen auszuwerten um auf Basis eines entsprechenden, durchschnittlichen Abstands große Ausreißer als Buchstaben zweier verschiedener Wörter zu identifizieren (angelehnt an [Par11], Seite 329 ff.).

Nachdem keine Kombination von Groß- und Kleinbuchstaben stattfindet, fällt zudem auf, dass aufgrund der gewählten Segmentierungs-Strategie (einzelne Zeichen werde immer auf die maximale Größe, unter Beibehaltung ihrer Proportionen, skaliert) eine Unterscheidung von Groß- und Kleinschreibung u.a. der Buchstaben i/I, s/S und o/O nicht möglich ist. Auch hier könnte die Einbeziehung des Gesamt-Bildes Abhilfe schaffen, indem die absolute Höhe der die einzelnen Zeichen umfassenden Rahmen für die Berechnung einer

entsprechenden Durchschnittshöhe herangezogen werden würde und anhand dieses Schwellwerts ein Maß an Zugehörigkeit zu einer Gruppe „Großbuchstaben“ (Höhe größer als die Durchschnittshöhe) bzw. „Kleinbuchstabe“ (Höhe kleiner als die Durchschnittshöhe) für betroffene Zeichen (welche bspw. ähnliche Wahrscheinlichkeiten für die Groß- und Kleindarstellung aufweisen) berechnet würde.

Inhalt der CD

Ordner-Struktur:

- dl4j
 - code
 - Java-Klassen der DL4J-Implementierungen
 - export
 - Diverse trainierte KNNs
- input
 - Prüf-Bilder (IBAN Handschrift, IBAN Maschinenschrift, Text Handschrift)
- MNIST
 - ubyte-Files und daraus generierte PNG-Dateien (gepackt)
- nnetwork
 - Java-Klassen der Eigenimplementierung
- opencv
 - code
 - GlyphExtraction.java
 - lib
 - opencv-310.jar und opencv_java310.dll (32- und 64-Bit)
- SD19
 - nist_sd19.zip (jeweils zufällig aus gesamten SD19-Bestand gewählte 2000 bzw. 4000 Trainings- und 1000 Testbilder, Grundlage der beschriebenen Messreihen)

Hinweis:

Die Ordnerbezeichnung orientiert sich an den im Text der Diplomarbeit genannten Projekt-Bezeichnungen.

In den entsprechenden Ordnern finden sich nur die Java-Klassen, etwaige Abhängigkeiten (OpenCV, DL4J, etc.) müssen selbstständig aufgelöst werden.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Telfs, den 28.9.2017

Thomas Pircher